

BUNDESREPUBLIK DEUTSCHLAND



Prioritätsbescheinigung über die Einreichung einer Patentanmeldung

Aktenzeichen:

199 52 587.0

Anmeldetag:

2. November 1999

Anmelder/Inhaber:

Gerd Krämer, Leimen/DE

Bezeichnung:

Verfahren zur Generierung einer einfachen Form künstlichen Bewußtseins im Computer zur Befähigung selbsttätig planender Erstellung von Maschinencode-Programmen und deren Ausführung zur Lösung beliebiger gestellter Programmieraufgaben

IPC:

G 06 F 9/44

Die angehefteten Stücke sind eine richtige und genaue Wiedergabe der ursprünglichen Unterlagen dieser Patentanmeldung.

München, den 22. Januar 2001
Deutsches Patent- und Markenamt
Der Präsident
Im Auftrag


Holß

Bezeichnung:

Verfahren zur Generierung einer einfachen Form künstlichen Bewußtseins im Computer zur
Befähigung selbsttätig planender Erstellung von Maschinencode-Programmen und deren
5 Ausführung zur Lösung beliebiger gestellter Programmieraufgaben

Inhalt:

| | |
|----|---|
| | 1. Beschreibung |
| 10 | 1.1 Aufgabenstellung und Stand der Technik |
| | 1.2 Herleitung der Realisierbarkeit und Definition künstlichen Bewußtseins |
| | 1.2.1 Philosophische Grundüberlegungen |
| | 1.2.2 Realisationsansatz zur Generierung von künstlichem Bewußtsein |
| | 1.3 technische Lehre zur Generierung künstlichen Bewußtseins |
| 15 | 1.3.0 Definition der verwendeten Abkürzungen |
| | 1.3.1 Verfahrensweise zur Generierung von künstlichem Bewußtsein mit einfachen Worten |
| | 1.3.2 Datenbank des KB-Wissens anlegen |
| | 1.3.2.x Beschreibungen der KB-Tabellen |
| 20 | 1.3.3 das System in den vorbereitenden Anfangszustand bringen |
| | 1.3.4 Basis-Lernen aus den Ausführungen aller OpCodes |
| | 1.3.4.1 OpCode generieren und ausführen |
| | 1.3.4.2 Analyse der OpCode-Auswirkung und Speichern der Ergebnisse |
| | 1.3.5 Realisierung der Grundbedürfnisse |
| 25 | 1.3.5.1 Realisierung künstlichen Schmerzes |
| | 1.3.5.2 Realisierung künstlichen Hungers |
| | 1.3.6 Planen nach den Kriterien des Bewertungssystems |
| | 1.3.7 Entwicklung des dynamischen Bewertungssystems |
| | 1.3.8 Selbstbewußtwerdung, Reproduktion und Evolution |
| 30 | 1.4 Programmieraufgabenstellung und Beispiele der Zielerreichung |
| | 1.4.1 Beispielaufg. 1: Erstellung eines Programms zur Berechnung des Mittelwerts |
| | 1.4.2 Beispielaufg. 2: Erstellung eines Programms zur Berechnung der Kubikwurzel |
| | 1.5 Festplatten-Speicherbedarf und Vergessen unwichtiger Daten |
| 35 | 1.5.1 Tabellengrößen |
| | 1.5.2 Vergessen |
| | 1.6 Bewußtwerdung |
| | 1.7 Darstellung der wirtschaftlichen Vorteile |
| | 2. Patentansprüche |
| | 3. Zeichnungen |
| 40 | 3.1 Relationale Datenbank des KB-Wissens |
| | 3.1.1 ER-Diagramm der KB-Datenbank: Fig. 1 |
| | 3.1.2 Tabellen der KB-Datenbank: |
| | Fig. 2-4: Identifikations- und Anfangsbedingungs-Tabellen: RIT, ICT, OIT |
| | Fig. 5-10: OpCode- und Kombinations-Tabellen: ORT, OLT, OBT; CxT(i) |
| 45 | Fig. 11, 12: Programmierzil-Tabellen: AST, ADT |
| | Fig. 13-15: Bewertungs-Tabellen: FIT, VFT |
| | Fig. 16: KB-Statuszeile: SAC |
| | Fig. 17-18: Energiespezifische Tabellen: ELT, EBT |
| | 3.2 Flußdiagramm des KB-Programms: |
| 50 | 3.2.1 Wertezuweisungen der OiT bzw. CxT(i): Fig. 19-21 |
| | 3.2.2 Wertezuweisungen der ELT und EBT: Fig. 22 |
| | 3.2.3 Definitionen zum Lesen des Flußdiagramms: Fig. 23 |
| | 3.2.4 KB-Flußdiagramm: Fig. 24a-d |

1. Beschreibung:

1.1 Aufgabenstellung und Stand der Technik:

- 5 Allein in Deutschland fehlen auf dem Gebiet der Software-Entwicklung z.Z. 80.000 Mitarbeiter und die Entwicklungs-Aufgaben werden immer komplexer.
Bislang werden Programme nach gegebener Aufgabenstellung von Software-Entwicklern konzipiert und programmiert.
Zur Erleichterung der Programmierung gibt es bislang "Wizards", die mittels Dialog-Fenstern nach interaktiver Eingabe von Daten des Benutzers grundlegende Teile des Source-Codes nach fest vorgegebenen Schemata erstellen.
- 10 Ferner werden auch firmenspezifische Skripte geschrieben, die mittels Auslesen von Daten aus ASCII-Files einfache stetig wiederkehrende Teile von Sourcecodes generieren.
In jedem Fall muß der Benutzer jedoch das erstellende Skript selbst schreiben und vorher die auszulesenden Daten generieren bzw. im Falle von "Wizards" vorher in Dialogfenstern benutzerdefinierte Eingaben machen und nach der Erstellung des Rahmen-Quellcodes die eigentliche Funktionsweise der Anwendung selbst ausprogrammieren. Danach muß noch der Source-Code in Maschinencode kompiliert werden, bevor er ausgeführt werden kann.
Lernfähig sind solche Programme jedoch nicht.
- 20 Auf dem Gebiet der KI gibt es neuronale Netzwerke / fuzzy Logic die zwar Expertensysteme bilden können, die äußere sensorische Reize aufnehmen und lernfähig in den "Neuronen" dann ihre Reaktion darauf entscheiden, also eine Art lernfähige Regelkreise bilden, jedoch können sie weder planen noch Maschinencode generieren und ausführen.
- 25 In 20 W (pat) 12/76 wurde die Generierung künstlichen Bewußtseins mit rein elektronischen Mitteln (Aneinanderreihung von Kameras und Monitoren) versucht - dieses Verfahren hier hat damit jedoch überhaupt nichts zu tun.
- 30 Mittels meinem Verfahren wird im Computer eine einfache Form von Bewußtsein erzeugt, das anfangs zwar ziellos willkürlich handelt, jedoch aus den Wirkungen seines eigenen "Verhaltens" lernt, um so, nachdem es die Wirkungen aller möglichen Handlungen kennt, später planend selektiv Einzelhandlungen aneinanderzuketten, z.B. um ein vorgegebenes Ziel zu erreichen.
Jeder Einzelhandlung entspricht hierbei eine Zahl, die, wenn man sie als OpCode (Maschinencode), also als direkte Prozessorsteuerungsanweisung, ausführt, entweder einen Fehler (Exception) erzeugt, oder eine Veränderung (z.B. der Registerinhalte) bewirkt.
Das System selektiert mittels geeigneter Analyse- und Bewertungsverfahren Codezusammenstellungen, die dann das gegebene Programmierziel erreichen.

40

1.2 Herleitung der Realisierbarkeit und Definition künstlichen Bewußtseins:

1.2.1 Philosophische Grundüberlegungen:

- 45 (... sind normalerweise kein Bestandteil einer Patentbeschreibung, jedoch für die Darlegung der Realisierbarkeit hier unerlässlich)

- Wäre die Voraussetzung des menschlichen Bewußtseins eine Art "Besetzung", die zwischen Zygote und Geburt stattfände, könnte man sie aufgrund von Gedankenexperimenten ungefähr lokalisieren:
- 50 Würde man gedanklich den Kopf abtrennen und die Halsschlagadern mit sauerstoff- und nährstoffreichen Blut versorgen, wäre das Bewußtsein sicher im Kopf.
Würde man nun das Gehirn bis auf die künstliche Versorgung völlig isolieren, wäre zwar auf herkömmliche Weise kein Informationsfluß zwischen dem Individuum und der Umwelt mehr möglich, das "Ich-Bewußtsein" wäre aber sicher noch vorhanden.
- 55 Jetzt kann man noch gedanklich die hinteren und seitlichen Großhirnlappen für Sehen, Hören, Riechen, Schmecken, Gleichgewicht, Sprache und das Kleinhirn abtrennen und es wäre nichts weiters verloren.

Bei den vorderen Hirnlappen verlöre man die Möglichkeit mit vorhandenem Wissen zu rechnen und bei einigen oberen Hirnteilen ginge Erinnerung verloren, aber das *Grund-Ich_bin* wäre immer noch vorhanden.

⇒ Wenn es eine Art "Seele" gäbe, läge sie am oberen Ende des Stamms.

5

Unter Berücksichtigung der gleitenden Evolution hätten dann Primaten aber auch eine "Seele". Und andere Säugetiere auch und alle anderen Tiere auch; und Einzeller auch; und Pflanzen auch.

⇒ Es fehlt eine "Bewußtseins-" bzw. "Beseelungsgrenze".

10 ⇒ Jede Zelle unseres Körpers müßte eine eigene Seele haben (die evolutionäre Spezialisierung zur Nervenzelle ist, wie auch bei den prenatalen Zellteilungen, fließend).

⇒ Es gibt keine "Seele".

⇒ Bewußtsein entsteht im Laufe der Evolution zwingend selbsttätig.

⇒ im "toten" Molekül der DNA liegt der Bauplan zur Generierung von Bewußtsein.

15 ⇒ Bewußtsein entsteht durch die Bewertung von eigenen Tätigkeiten und deren Auswirkungen, mit der Reflexion der Bewertungsergebnisse auf das sich anpassende Bewertungssystem.

⇒ Wenn zur Generierung von Bewußtsein keine "Seele" notwendig ist, sondern nur das komplexe "Programm" der DNA, dann ist Bewußtsein auch durch ein komplexes reflexives Computer-Programm generierbar.

20

1.2.2 Realisationsansatz zur Generierung von künstlichem Bewußtsein:

Das Handeln aller, auch der einfachsten Individuen dient zur Erfüllung ihrer Grundbedürfnisse.

25 Diese Grundbedürfnisse sind:

a.) kein Schmerz := kein Angriff auf's System und

b.) kein Hunger := kein drohender Energieverlust

Ein komplexes Programm, in dem diese Grundbedürfnisse modelliert sind, und das frei Handeln kann und in der Lage ist, wie ein Kind aus seinen Handlungen reflexiv zu lernen, was seine Handlungen bewirken und ob seine Handlungen seine Situation im Bezug auf seine Grundbedürfnisse verbessern oder verschlechtern, baut ein Bewertungssystem auf, plant dann seine Handlungen aus dem Erlernten zur Verbesserung seiner Situation und erlangt so Bewußtsein.

30 Scannt es auch einmal seinen eigenen Code, probiert aus, was seine Code-Abschnitte bewirken und erkennt deren Zusammenhang, erlangt es sogar Selbst-Bewußtsein, und kann nicht nur seinen Code reproduzieren, sondern seinen Code bei der Reproduktion auch bewußt verändern und verbessern, also eine Evolution aufgrund seiner Erkenntnisse beginnen.

40

1.3. technische Lehre zur Generierung künstlichen Bewußtseins:

1.3.0 Definition der verwendeten Abkürzungen:

Die Programmierung funktioniert auf jedem Computer mit jedem beliebigem Prozessor und jedem beliebigem Betriebssystem. Die mit μ indizierten Abkürzungen sind für die Motorola-Prozessoren MC680x0 spezifisch, die mit π indizierten für die Intel-Pentiums und ψ -indizierte kennzeichnen den PowerPC-RISC-Prozessor:

- äq. = äquivalent(e(s))
- 50 ASR $_{\psi}$ = Adress Space Register
- BAT $_{\psi}$ = BAT-Registers
- BC = BitCode: jedes Bit entspricht einem Code und es sind Codekombinationen zulässig.
- CCR $_{\mu}$ = Condition-Code-Register (= Flags: EXtension, Negativ-, Zero-, Overflow-, Carry-)
- CISC = Complex-InstructionSet Computer (z.B. IA $_{\pi}$ und MC $_{\psi}$)
- 55 CPU = Central processing Unit = Prozessor
- CR $_{\psi}$ = Condition-Register (CR 0..7)
- CR $_{\pi}$ = Control-Register

| | | |
|----|---------------------|--|
| | CTR _v | = Count-Register |
| | DABR _v | = Data Address Breakpoint Register |
| | DAR _v | = Data Address Register |
| | DB | = DatenBank |
| 5 | DEC _v | = Decrement-Register |
| | DR _π | = Debug-Register |
| | DSISR _v | = DSI Status-Register zeigt den Grund für DSI- und Alignment-Exceptions an. |
| | EA | = Effective Address (direkter Speicherzugriff ohne Register) |
| | EAR _v | = External Access Register |
| 10 | Rseg _π | = Segment Register: CS; SS; DS, ES, FS, GS |
| | EFlags _π | = 32-Bit Register mit den System-Flags: <u>IDent</u> -, <u>VirtualInterruptPending</u> -, <u>VirtualInterruptFlag</u> -, <u>AlignmentCheck</u> -, <u>Virtual8086Mode</u> -, <u>ResumeFlag</u> -, <u>NestedTask</u> -, <u>InputOutputPrivilegeLevel</u> -, <u>OverflowFlag</u> -, <u>DirectionFlag</u> -, <u>InterruptEnableFlag</u> -, <u>TrapFlag</u> -, <u>SignFlag</u> -, <u>ZeroFlag</u> -, <u>Auxiliary/Align-CarryFlag</u> -, <u>ParityFlag</u> -, <u>CarryFlag</u> . |
| 15 | EIP _π | = Extended Instruction Pointer (\triangleq PC _μ) |
| | ER | = Entity Relationship (Datenbankmodell) |
| | ESP _π | = Extended StackPointer (\triangleq USP _μ) |
| | Exc. | = <u>Exception</u> _π : # <u>DeviceError</u> , # <u>DeBug</u> , NMI IRQ, # <u>BreakPoint</u> , # <u>OverFlow</u> , # <u>BoundRange</u> exceeded, # <u>UD</u> (Invalid Opcode), # <u>NM</u> (device not available), # <u>DoubleFault</u> , invalid # <u>TaskSwitch</u> , Segment # <u>NotPresent</u> , # <u>SS</u> (StackFault), # <u>GeneralProtection</u> , # <u>PageFault</u> , # <u>MF</u> (FloatingPoint-Error), # <u>AlignmentCheck</u> , # <u>MachineCheck</u> . |
| 20 | | <u>Exception</u> _μ : Reset, BusError, AddressError, invalidOpCode, Div/O, CHK, TrapV, PrivilegeViolation, Trace, Interrupts, Traps. |
| | | <u>Exception</u> _v : System-Reset, Machine-Check, DSI, ISI, Ext.Interrupt, Alignment, Program, Floating-Point unavailable, Decrementer, System Call, Trace, Floating-Point Assist. |
| 25 | FFT | = fast Fourier-Transformation |
| | FK | = Foreign Key einer ER-Datenbanktabelle |
| | FPR _v | = Floating-Point Register 0..31 |
| 30 | FPSCR _v | = Floating-Point Status and Control Register |
| | GB | = GigaBytes = 2 ³⁰ Bytes |
| | GPR | = General Purpose Registers: beim Pentium _π : EAX, EBX, ECX, EDX; ESI, EDI, EBP; ESP; EIP und beim PowerPC _v : GPR 0..31 und bei Motorola _μ die Daten- und Adress-Reg. |
| | IA | = Intel-Architecture |
| 35 | IRQ | = Interrupt-Request |
| | KB | = Künstliches Bewußtsein |
| | kB | = KiloBytes = 2 ¹⁰ Bytes |
| | ld | = Logarithmus dualis = log ₂ |
| | LR _v | = Link-Register |
| 40 | MB | = MegaBytes = 2 ²⁰ Bytes |
| | MSR _π | = Model Specific Register |
| | MSR _v | = Machine State Register |
| | NMI | = Non-Maskable-Interrupt (höchster Interrupt) |
| | NOP | = NoOperation-OpCode [Maschinencodebefehl ohne Wirkung (außer IP _π /PC _μ ++)] |
| 45 | OLB | = OpCode Lower Byte = letztes Byte im OpCode |
| | PC _μ | = Programm-Counter (=Pointer auf 1.Byte der Speicherstelle, an der sich der Prozessor im Programm gerade befindet, vor der dortigen OpCode-Ausführung) |
| | PK | = Primary Key einer ER-Datenbanktabelle |
| | PVR _v | = Processor Version Register |
| 50 | RISC | = Reduced-InstructionSet Computer (z.B. PowerPC _v) |
| | RTE _μ | = Befehl: Return from Exception (lädt SR und PC vom Supervisor-Stack) |
| | SDR1 _v | = SDR1-Register |
| | SPRG _v | = SPRG 0..3 |
| | SR _μ | = Statusregister (Flags _μ : <u>Trace</u> -, <u>Supervisor</u> -, + Interrupt-Maske: I ₂ I ₁ I ₀ . + CCR-Flags) |
| 55 | SR _π | = Segment Registers: CS, DS, SS, ES, FS, GS |
| | SR _v | = Segment Registers |
| | SRR _v | = Save/Restore-Register of Machine-Status |

- SSP_{μ} = Supervisor-Stackpointer (A7 im Supervisor-Modus)
 TB_{π} = Time Base Facility: Time-Counter $\rightarrow 2^{64}-1$
 TR_{π} = Table-Register: GDTR, IDTR, LDTR, TR
 USP_{μ} = User-Stackpointer (Adressregister A7 im User-Modus, A7' im Supervisor-Modus)
 5 \triangle = entspricht
 $\$$ = Beginn einer Hexadezimalzahl
 ζ = Ergebnis des bitweisen AND über alle folgenden Werte $[=V_1 \& V_2 \& \dots \& V_n]$
 Ξ = Ergebnis des bitweisen OR über alle folgenden Werte $[=V_1 | V_2 | \dots | V_n]$
 γ = Anzahl der gesetzten Bits im folgenden Wert $[=(1\&V) + (2\&V)/2 + (4\&V)/4 + \dots]$
 10 \forall = für alle anderen ... (nur $\forall \triangle$ für alle ...)

1.3.1 Verfahrensweise zur Generierung von künstlichem Bewußtsein mit einfachen Worten:

- 15 Ein Computersystem erlangt Bewußtsein, wenn das aktive Programm, bei dem alle Exception-Vektoren abgefangen sind, und Grundbedürfnisse modelliert sind, folgendermaßen verfährt:
 Generiere eine Zahl und verwende sie als OpCode (=Operation-Code=Maschinencode-Befehl); führe ihn aus, werte aus, was er bewirkt hat und speichere die ermittelte Wirkung der Ausführung. Fahre so mit allen Zahlen \rightarrow OpCodes mit vielen repräsentativen Anfangsbedingungen
 20 (Register-Werte und Adressregister-Verweisinhalte) fort.
 Benutze dann so die gespeicherten OpCodes, die keine oder nur selten eine Exception verursachen und werte aus, ob deren Ausführung die eigenen Grundbedürfnisse erfüllen, oder ihnen abträglich sind.
 Kette dann die OpCodes aneinander, die die eigene Situation nicht verschlechterten, und werte
 25 die Wirkung dieser Code-Kombinationen aus, und speichere deren Wirkung.
 Plane so Codekombinationen, die das Wohlbefinden bzgl. der Grundbedürfnisse (Modellierung siehe 1.3.5) erhöhen bzw. für das gegebene Programmierzil von Bedeutung sein könnten.

30 1.3.2 Datenbank des KB-Wissens anlegen:

- Damit das Erlernte des KB-Programms persistent bleibt und die großen Datenmengen komfortabel erreichbar sind, wird eine relationale Datenbank mit den Tabellen und deren Relationen gemäß 3.1 angelegt; zwecks Zugriffsbeschleunigung und Speicherplatzersparnis äquivalente
 35 PKs als Cluster, und es werden zwecks Erhöhung der Zugriffsgeschwindigkeit für benötigte Non-PK-Spalten weitere Indexe angelegt. Das ER-Diagramm zeigt Fig.1.
 Je nach Prozessor und wieviele 32-Bit-Befehle dieser hat, kann die Datenbank sehr groß und die Zugriffsgeschwindigkeiten entsprechend langsam werden. Deshalb eignen sich RISC-Prozessoren eher für das KB-Programm als CISC-Rechner. Aber auch die CISC-Maschinen, wie die der
 40 IA, benutzen nur bei relativ wenigen Befehlen mehr als 24-Bit, weshalb man mit über mehrere Festplatten gestribeten Tabellen und zusätzlichen Index-Festplatten und mit erhöhtem "Vergessen" bei ineffizienten Befehlskombinationen ebenfalls sehr gut arbeiten kann.
 Auf das Speicherplatzproblem wird unter 1.5 eingegangen.

45 1.3.2.1 Die Register-Identifikations-Tabelle [RIT - Fig.2]:

- In der RIT werden die Daten jedes Prozessor-Registers initial eingegeben: Jedes Register enthält eine Identifikations-Nummer ein zugewiesenes Bit im BitCode ein Zeichen zur Beschreibung des Register-Typs, eine laufende Nummer dieses Register-Typs und optional eine Beschreibung des Registers. Das Register der Flags ($EFlags_{\pi}/SR_{\mu}$) hat die Register-ID 0. Die Exception-Vektoren
 50 sind zwar meist keine Register, sondern direkte Adressen im Speicher - um diese wichtigen Vektoren ebenfalls identifizieren zu können, erhalten sie Redister-IDs mit negativem Vorzeichen, die der Exception-Vektornummer entsprechen [wenn Exception-Nr. 0 nicht Reset, sondern eine echte Exception ist, um 1 verschoben - dann: $Register_ID = -(ExceptionNr + 1)$].
 In Fig.2b ist für das Beispiel Motorola dargelegt, wie die RIT aussehen könnte.

1.3.2.2 Die Operations-Identifikations-Tabelle (OIT - Fig.4):

Wie in der RIT die Register, bekommen in der OIT die wichtigsten Operationen eine Identifikationsnummer und ein Bit im BitCode zugeordnet.

Der *Operation_Type* ordnet die Operation in Gruppen ein, die in Fig.4c beschrieben sind.

- 5 Die *Operation_Mnemonic* (braucht nicht exakt zu sein) und die optionale *Operation_Description* beschreiben, um was für einen Befehl es sich handelt.

1.3.2.3 Die Anfangsbedingungen-Tabelle (ICT (initial conditions) - Fig.3):

- 10 Da für gleiche OpCode-Ausführungen, je nach Anfangsbedingungen, unterschiedliche Wirkungen auftreten können, werden in dieser Tabelle repräsentative Anfangsbedingungen vorgegeben. Für jede Anfangsbedingungsnummer (hier -31...+30) werden für alle positiven *Register_IDs* ein Sample von Anfangsbedingungen z.B. nach der in Fig.3b vorgestellten Funktion generiert.

Jedoch nur für alle Register, die mathematisch verwertbare Zahlen enthalten können, wie Daten-Register, Adress-Register-Verweise und die Adressierung darunter (wg. -(Adr.Reg.)),

- 15 Floating-Point-Register und sonstige spezielle Rechen-Register (z.B. MMX).

Mit Adress-Registern läßt sich zwar auch rechnen, jedoch haben sie immer Werte die an Speicheradressen verweisen, an denen dann die vordefinierten Verweis-Inhalte stehen. Somit können die Anfangsbedingungen für die Adressregister allenfalls zyklisch durch die Testwerte in den Verweisen laufen.

- 20 Das Status-/EFlags-Register hat in den höheren Bytes immer die gleichen Anfangswerte aus *SAC.actual_Processor_Mode*. Bei den ConditionCodes im untersten Byte können jedoch die Anfangswerte variieren. Mit den Control-, Debug- und Maschinenstatus-Registern, sowie sonstigen Spezialregistern wird anfangs ebenfalls kein Unfug getrieben und sie werden immer auf die gleichen, Default-Werte gesetzt.

25

1.3.2.4 Die OpCode-Register-Tabelle (ORT - Fig.5):

Für jedes durch die OpCode-Ausführung veränderte Register der aktuellen Anfangsbedingungen wird in einer Schleife über alle möglichen Quellregister ermittelt, durch welche mögliche Operation mit welchem Quellregister der Wert im Zielregister entstanden sein könnte. Für jede zutreffende Quell-Ziel-Register-Kombination wird ein Tabelleneintrag generiert (bei unitären Operationen ist *Register_ID_source* = -1) und für jede zutreffende Operationsmöglichkeit das zugehörige Bit entsprechend der OIT im *Operations_BitCode* gesetzt. Wie alle Felder der ORT berechnet werden, ist in Fig.5 beschrieben. Fig.19 enthält die Wertezuweisungs-Algorithmen.

35 1.3.2.5 Die OpCode-Lern-Tabelle (OLT - Fig.6):

Die OLT stellt eine Zusammenfassung der Auswirkungen des aktuellen OpCodes unter den verschiedenen verwendeten Anfangsbedingungen dar.

In den ersten 6 Spalten werden Informationen über fatale Auswirkungen dieses OpCodes gesammelt. Dann kommt die Differenz des Programmzählers nach der OpCode-Ausführung zum

- 40 Wert davor und nun die Condition-Codes, die einen Sprung ausgelöst haben könnten (redundant *ICT.Register_Value(Register_ID=0)*).

Danach kommen in *Register_changed_BitCode* und *Register_source_BitCode* die Bits aller möglichen Ziel- und Quell-Register aus den zugehörigen ORT-Einträgen und in *max_Operation_BitCode* und *min_Operation_BitCode* die bitweise geOReten und geANDeten BitCodes der

- 45 ORT.*Operations_BitCode* -Einträge.

Die Dauer und der Zeitpunkt der OpCode-Ausführung werden gespeichert und in *aim_valuation* wird entsprechend *VFT.Valuation_Function(ADT.aim_Valuation_FunctionID)* bewertet, wie wertvoll der OpCode unter diesen Anfangsbedingungen für die Programmierzilerreichung war (Wertezuweisungen nach Fig.20).

50

1.3.2.6 Die OpCode-Basis-Tabelle (OBT - Fig.7):

Die OpCode-Basis-Tabelle beschreibt die ermittelte Gesamtwirkung eines OpCodes unter allen verwendeten Anfangsbedingungen. In Fig.21 ist dargelegt, wie die Auswertung (Füllen der Spalten) erfolgt, um einen "Steckbrief" des OpCodes zu erstellen.

- 55 Die OBT enthält das Resummé aller Ausführungen dieses OpCodes und ist später beim zielgerichteten planen von Codekombinationen wichtig.

1.3.2.7 Die Kombinations-Register-Tabellen [CRT(i) - Fig.8]:

Die Kombinations-Register-Tabellen werden dynamisch angelegt und entsprechen der der ORT, mit dem Unterschied, daß hier die Wirkungen von OpCode-Kombinationen analysiert werden. CBT(1)=OBT, CBT(2) hat einen OpCode mehr im PK, CBT(3) hat 3 OpCodes im PK, u.s.w.

5

1.3.2.8 Die Kombinations-Lern-Tabellen [CLT(i) - Fig.9]:

Hier gilt das gleiche analog der CRT(i). Die CLT(i) geben die Wirkung der OpCode-Kombination bei den verwendeten Anfangsbedingungen wieder. Jetzt gewinnt auch das Feld *CLT(i).gradient_aim_valuation* an Bedeutung. Während es noch bei der CLT(1)=OLT identisch *aim_valuation* ist, enthält es bei den CLT(i≥2): *CLT(i).aim_valuation - CLT(i-1).aim_valuation* (Fig.20 unten).

10

1.3.2.9 Die Kombinations-Basis-Tabellen [CLT(i) - Fig.10]:

Die CBT(i) geben folglich das Resummé der Wirkung der OpCode-Kombination wieder. Die Wertezuweisungen erfolgen nach Fig.21. Die gerade höchste CBT(n) ist die CPT (Kombinations-Plan-Tabelle = Entstehungsort des Lösungsprogramms).

15

Ergibt *ADT.aim_fulfilled_Flag_Function(CPT-PK)=1 (TRUE)*, wurde gerade ein Lösungsprogramm der gestellten Programmieraufgabe gefunden, und es wird in der AST eingetragen.

1.3.2.10 Die Ziellösungs-Tabelle [AST (aim solution) - Fig.11]:

Die AST enthält für jede gestellte Programmieraufgabe die Lösungsprogramme, deren Programmlängen, die Ausführungszeiten und die je benutzten Register und Operationen.

20

1.3.2.11 Die Ziel-Beschreibungs-Tabelle [ADT (aim description) - Fig.12]:

Die ADT ordnet jedem Programmierzil eine Identifikationsnummer zu, eine Kurzbeschreibung, die BitCode-Kombinationen der zu verwendenden Quell- und Ziel-Register, die Register und Operationen, die im Lösungsprogramm nicht verwendet werden sollen, frühere Lösungsprogramme, die mitverwendet werden könnten und eine Funktion, die TRUE ausgibt, wenn die OpCode-Kombination für die Ein- und AusgabeRegister eine Lösung der gestellten Aufgabe darstellt, sowie eine Ziel-Annäherungsfunktion, die die Zielnähe der akt. OpCode-Kombination (=CPT-PK) bewertet.

30

1.3.2.12 Die Funktion-Identifikations-Tabelle [FIT - Fig.13,14]:

In der FIT werden Teilfunktionen, die für die Zusammenstellung der Energie-Bewertungsfunktion verwendet werden könnten, zur Verfügung gestellt.

35

Sie wird in 2 Variationen vorgestellt:

a.) für die Erstellung einer dynamischen Bewertungsfunktion in SQL,

b.) für die Erstellung einer dynamischen Bewertungsfunktion in Maschinensprache.

Der veränderliche Aufbau einer Bewertungsfunktion ist in SQL viel einfacher zu bewerkstelligen, jedoch sind die Ausführungszeiten entsprechend langsam und es muß nach jeder Zusammenstellung neu geparsed werden.

40

Zukünftig soll auch die Bewertungsfunktion gleich in Maschinencode zusammengestellt werden. Das auch hat den Vorteil, daß das KB-Programm manche gelöste Programmierziele als FIT-Teilfunktionen für die Zusammenstellung der Bewertungsfunktion weiterverwenden kann.

45

1.3.2.13 Die Bewertungs-Funktions-Tabelle [VFT (valuation f.) - Fig.15]:

In der VFT liegt das dynamische Bewertungssystem im Bezug auf das eigene "Wohlbefinden" (Energie-Register) und der Programmierzelnähe.

Die *VFT.Valuation_Function(Type='E', SAC.Energy_Valuation_Function_ID)* bewertet energie-spezifische Handlungen und die *Valuation_Function(Type='A', SAC.Aim_Valuation_FunctionID)* die Programmierzil-Erreichungsnähe.

50

Die *VFT.Function_ID_Chain* enthält die Verkettung der Function-ID's, also die Teilfunktions-Ausführungs-Reihenfolge: Hier bewirkt NUM, daß der nächste Wert eine Byte-Zahl ist, VALUE, daß der nächste Wert die LfdNr der CLT(n)-Spalte ist, der ein Wert entnommen wird, EREG die Register_ID des Energie-Registers, S/D REG der Wert aus der *ADT.all_source/dest_Registers_BitCode* und AIM_F das Ergebnis aus der *ADT.aim_fulfilled_Flag_Function*. Die unitären Operationen wirken auf das letzte Ergebnis und die binären auf die letzten 2 Ergebnisse aus der *Function_ID_Chain*.

55

Bei jeder Anpassung, Erweiterung oder sonstigen Verbesserung dieser Bewertungsfunktionen

wird die *Valuation_Function_ID* incrementiert und ein neuer Eintrag mit der veränderten *Valuation_Function* generiert und alle Bewertungsfunktionen in ihrer Effizienz neu bewertet: $VFT.Valuation_Function_value = SAC.Energy/Aim_self_valuation_Func(\dots)$, um einen Effizienz-Gradienten als Referenz für weitere Verbesserungen zu haben.

- 5 Die Funktionsweise des dynamischen Bewertungssystems ist unter 1.3.7 beschrieben.

1.3.2.14 Die Statuszeile des KB-Programms [SAC (state artificial consciousness) - Fig. 16]:

Diese Tabelle hat keinen Key und nur einen Eintrag. Er enthält die Statusinformationen des KB-Programms und zwei Selbstbewertungs-Funktionen, die Effizienz der Energie- und der Zielannäherungs-Bewertungsfunktionen der VFT anhand ihrer gelieferten Ergebnisse bewertet. Diese Selbstbewertungsfunktionen werden, im Gegensatz zur Energie- und Zielnähe-Bewertungsfunktion, vom Programm selbst nicht mehr verändert, können jedoch vom Benutzer angepaßt werden.

10 **1.3.2.15 Die Energie-Lern-Tabelle [ELT - Fig. 17]:**

In der ELT werden Daten über alle energiespezifischen Handlungen der akt. Anfangsbedingungen, also über alle OpCodes und Code-Kombinationen, die das letzte Datenregister betreffen, gesammelt.

Die Wertvolligkeit der energiespezifischen Handlung wird nach $ELT.Energy_valuation =$

- 20 $VFT.Valuation_Function(SAC.Energy_Valuation_Func_ID)$ bewertet.

1.3.2.16 Die Energie-Basis-Tabelle [EBT - Fig. 18]:

Ähnlich wie in den CBT(i), werden in der EBT die Auswirkungen einer energieändernden Code-Kombinationen, als Resummé aller Anfangsbedingungen, gesammelt.

25

1.3.3 Das System in den vorbereitenden Anfangszustand bringen:

Um das System später wieder ohne neues booten in den ursprünglichen Zustand versetzen zu können, müssen einige Pointer (= Zeiger = Adressen) gesichert (zwischengespeichert) werden. Danach werden die Exception-Vektoren mit eigenen Abfang-Routinen belegt, da das System anfangs Zahlen willkürlich als Maschinencode generiert und ausführt, obwohl viele dieser Zahlen als OpCode unzulässig sind oder aufgrund der gerade gewählten Anfangsbedingungen Fehler verursachen. Systemabstürze wären die Folge, wenn man nicht alle Exception-Vektoren abfinge.

35

Will man das bewußtseingenerierende Programm laufen lassen, muß man also, im Falle, daß es als einziges Programm im Computer laufen soll:

- 40 a.) das Multitasking abschalten, in dem man dieses entweder mit einer Betriebssystemroutine disabled oder indem man die IRQ-Maske des Prozessors auf NMI setzt.
b.) alle System-Exception-Vektoren sichern.
c.) alle System-Exception-Vektoren des Prozessors auf eigene Behandlungsroutinen umlenken, oder wenn man es später einmal neben anderen Programmen und vielleicht auch weiteren k.B.-Programmen parallel laufen lassen will:
45 a') die eigene Task-Priorität etwas erhöhen.
b') alle Task-Exception-Vektoren sichern.
c') alle Task-Exception-Vektoren des KB-Programms auf eigene Behandlungsroutinen umlenken.
d.) das Statusregister, ($\Delta EFlags_{\mu}$) und den User-Stackpointer sichern.
50 e.) die Werte der anderen Adressregister und die der Datenregister sichern.
f.) die Werte der Segment-, Control-, Debug- und Spezialregister sichern.
g.) manche Exception-Vektoren auf besondere Abfangroutine setzen, die berücksichtigt, daß zusätzliche Daten (z.B. bei Adressfehler: zusätzlich Zugriffsadresse + Opcode) auf den Supervisor-Stack geladen werden.
55 h.) Privilege-Violation-Exception-Vector auf besondere Abfangroutine setzen.
i.) einen Trap-Vektor auf eine Routine setzen, bei der im Supervisor-Modus fortgeführt werden soll.

- j.) diesen Trap ausführen (CPU schaltet sich jetzt in den Supervisor-Modus und macht ab dieser Trap-Vektoradresse weiter).
 - k.) Trace-Exception-Vektor auf eigene Trace-Routine zur Wirkungs-Analyse setzen.
 - l.) Flags des ersten Word auf dem Supervisor-Stack so setzen, daß beim Laden des SR vom SSP das Trace-Flag und die IRQ-Maske \rightarrow NMI gesetzt werden (bei Motorola ist das #8700), denn während des folgenden Basis-Lernens soll noch kein Interrupt möglich sein.
- Siehe hierzu Fig. 24a.

10 1.3.4 Basis-Lernen aus den Ausführungen aller OpCodes:

1.3.4.1 OpCode generieren und ausführen:

- a.) 32-Bit-Zahl als OpCode generieren, angefangen bei \$0000.0000, im weiteren Verlauf immer um 1 hochzählen [dabei kann man von vorn herein die OpCodes überspringen, die offensichtlich Unfug machen würden (siehe BitCodes des CPU-Befehlssatzes)].
- 15 b.) Daten- und Adress-Register, sowie die AdressRegister-Verweisinhalte und die Verweisinhalte ein DWord darunter auf vordefinierte Testwerte laden und die ConditionCodes in EFlags_μ/CR_μ löschen.
- c.) Den generierten OpCode an die Teststelle im Speicher schreiben. Hinter der Teststelle muß noch mit ausreichend NOP-OpCodes (oder besser mit Nullen, wenn diese der Mnemonic "ORI #0, Reg.0" entsprechen) aufgefüllt werden, da es sich um einen langen Befehl handeln könnte und auch der Fall der Trace-Bit-Löschung mitberücksichtigt werden muß (dahinter steht die Trace-Bit-Cleared Abfangroutine).
- 20 d.) Supervisor-Stack-Inhalt so setzen, daß beim Rücksprung aus dem Supervisor-Modus das Statusregister mit gelöschtem CCR, IRQ-Maske auf NMI, Trace-Bit gesetzt und Supervisor-Bit(Maske) gelöscht, geladen wird und das dahinter befindliche Langwort die Test-Adresse darstellt. Rücksprung aus Supervisor-Modus (RTE_μ) ausführen: das EFlags_μ/Status-Register_μ wird mit o.g. Werten initiiert und der IP_μ/PC_μ mit der Testadresse geladen und der an der Teststelle befindliche OpCode ausgeführt.
- 25 • Ist jetzt eine Exception (außer Trace) aufgetreten, wird die Exception kurz grafisch angezeigt und bei der Generierung des nächsten OpCodes fortgeführt. Diesen OpCode kann das Individuum vergessen. [Achtung: bei manchen Exceptions tritt wegen Trace eine Kombination des Exception-Handlings auf.]
- 30 • Tritt weder Trace, noch eine andere Exception auf, wurde das Trace-Bit gelöscht (sollte bei Einzel-OpCodes nie auftreten) und die hinter der Teststelle befindliche Abfangroutine wird ausgeführt.
- 35 • Bei Trace-Exception (Normalfall) wurde ein benutzbarer OpCode generiert, dessen Ausführungs-Auswirkungen jetzt analysiert werden müssen.

40 1.3.4.2 Analyse der OpCode-Auswirkung und Speichern der Ergebnisse:

- a.) Das EFlags_μ/Status-Register_μ und die Daten- und Adressregister, sowie Adressregister-Verweisinhalte und die Verweisinhalte des DWords vor den Adressregistern und den User-Stackpointer zur Analyse speichern.
- 45 b.) Überprüfung der eigenen Maschinencode-Checksum des aktiven KB-Programms und der CheckSum der inaktiven Kopie im RAM (je Test-OpCode-Speicherstelle): Bei CheckSum-Änderung hat sich das Programm bei der Ausführung "verletzt" (Programmteile selbst überschrieben). Dann das entsprechende Corrupt-Flag in der Tabelle setzen. Bei Verletzung der aktiven Version in die inaktive Kopien springen, dann den Code vergleichen und die korrupten Bytes durch Code der unverletzten Version ersetzen.
- 50 c.) Die Supervisor-BitMaske des gesicherten User-Stackpointers auf Stack prüfen: War der Prozessor vor Ausführung der Exception im Supervisor-Modus (obwohl er bei der Test-OpCode-Ausführung im User-Modus war), ist eine Kombination von der normalen Trace-Exception mit einer niedrig priorisierten Exception aufgetreten [z.B. bei Motorola Div/0-, Trap- oder Chk-Exception (im 68000er Handbuch nicht dokumentiert)]. D.h. der Prozessor holte erst den Exception-Vektor des gerade aufgetretenen niedriger priorisierten Fehlers und lud das Statusregister und den Programmzähler auf den Supervisor-Stack und sprang dann noch während dieser prozessorinternen Routine, noch vor Beginn der Exception-Routine in
- 55

die weitere Trace-Exception-Routine, wodurch wieder Programmzähler und das Status-Register auf den Supervisor-Stack geladen wurden.

Mittels der auf dem Supervisor-Stack gesicherten Supervisor-Bits des Statusregisters ist nun feststellbar, daß vor Trace bereits eine Exception auftrat und durch Vergleich des zweiten
5 auf dem Supervisor-Stack gesicherten Programmzählers mit den niedrigpriorisierten Exception-Vektoren ist nun die ursprüngliche Exception vor Trace ermittelbar, deren Exception-Nummer abgespeichert wird.

d.) Vergleich des IP_{π}/PC_{μ} auf dem Supervisor-Stack mit der Test-OpCode-Adresse: Wurde der IP_{π}/PC_{μ} erniedrigt, blieb unverändert oder um einen Wert größer als der längstmögliche
10 OpCode erhöht, war es ein Sprung.

Wurde er um mehr als 4 Bytes erhöht, war es ein langer Befehl oder ein kurzer Vorwärtssprung - die Differenzierung ergibt dann daraus, ob Register verändert wurden. Dieses Analyse-Ergebnis wird wieder abgespeichert.

e.) Beim Vergleich des $EFlags_{\pi}/Status-Registers_{\mu}$ und aller Registerinhalte und der AdressRegister-Verweisinhalte, und der AdressRegister-Verweisinhalte einer max. Adressierbarkeitslänge
15 darunter (wg. -(Adr.Reg.)) mit den Original-Werten.

In einer BitMaske wird nun geflaggt, welche Register geändert wurden und es wird analysiert, welche Operationen mit welchen Quell- und Zielregistern stattgefunden haben könnten (hierbei Änderungen des $EFlags_{\pi}/SR_{\mu}$ beachten) und das Ergebnis wird in der ORT und OLT
20 gespeichert (siehe Figs.5,19;6,20) und die OBT aktualisiert (Fig.7,21).

f.) Bei Sprüngen Analyse des $EFlags_{\pi}/SR_{\mu}$, ob der Sprung bedingungsabhängig war.

1.3.5 Realisierung der Grundbedürfnisse:

25

1.3.5.1 Realisierung künstlichen Schmerzes:

Schmerz wird durch die Verletzung des Systems verursacht. Der ursprünglichste Schmerz in der biologischen Evolution kommt bereits beim Einzeller vor und ist die Verletzung der DNA im Zellkern. Ist die DNA verletzt, muß sich der Einzeller unter Aufbringung seiner Ressourcen die
30 Zeit nehmen, seine DNA zu reparieren. Er tut dies, indem er die fehlenden Aminosäuren in der defekten Doppelhelixhälfte durch komplementäre Basen der intakten Doppelhelixhälfte komplementär ersetzt.

Das KB-Programm wird doppelt in's RAM geladen. Führt das KB-Prg. (oder ein anderes) einen Befehl aus, der seinen aktiven oder inaktiven Code im Speicher überschreibt, wird es also in der
35 aktiven oder inaktiven Form verletzt, kann es diese Verletzung anhand einer Änderung seiner CheckSum erkennen und muß sich nun die Zeit nehmen, bei Verletzung des aktiven Codes nun in den bisher inaktiven äquivalenten Code zu springen und dann beide Codes 32-Bit weise vergleichen und an der Stelle der Ungleichheit das DoubleWord seines Codes mit unveränderter CheckSum an die verletzte Stelle des Codes mit veränderter CheckSum schreiben, um sich zu
40 heilen.

1.3.5.2 Realisierung künstlichen Hungers:

Hunger ist drohender Energieverlust. Energie wird in den Zellen u.a. durch Umwandlung von Adenosintriphosphat in Adenosindiphosphat erzeugt. Die Energie zum Aufbau von Adenosintriphosphat aus Adenosindiphosphat wird durch Verbrennung von Glucose gewonnen. Fehlende
45 Energie macht in den Zellen den Stoffwechsel und somit jede Handlung, Reaktion auf Schmerz oder die Selbstheilung bei Verletzung unmöglich.

Die "Energienmenge" des KB-Programms läßt sich durch die Höhe eines Werts in einem Datenregister modellieren. Es wäre nun möglich, Hunger durch abnehmende Stromversorgung
50 des Prozessors durch externes Auslesen dieses Datenregisters und Erhöhung eines Ohmschen Widerstandswerts der Prozessorstromversorgung zu realisieren. Eine weniger authentische, hardwareungebundene Lösung ist auch möglich:

Fehlende Energie ist dem Lernprozess abträglich. Bei mäßigen Werten in o.g. Datenregister treten Fehler beim Lernen aus den OpCode-Ausführungen auf. Bei geringen Werten ist das
55 Lernen aus OpCode-Ausführungen nicht mehr möglich und kleinste Werte in diesen Datenregister lassen gespeichertes Wissen vergessen. Ist der Wert auf null tritt zusätzlich Schmerz, also EigenCode-Verletzung auf.

Das KB-Programm muß also bei Hunger OpCodes finden und ausführen, die den Wert dieses Datenregisters erhöhen.
Die abnehmende Energie, also das Entstehen von Hunger, wird dadurch simuliert, daß das KB-Programm selbst nach jeder Handlung (= OpCode-Ausführung) dieses Register um 1 erniedrigt.

5

1.3.6 Planen nach den Kriterien des Bewertungssystems:

- Hat das Individuum alle ihm möglichen OpCodes getestet und sich die Auswirkungen der verwendbaren Befehle gemerkt, kann es aufgrund seines Wissens zur Befriedigung seiner Grundbedürfnisse und zur Erreichung von Programmierzwecken nun lernen zu planen:
- 10 Hierfür reiht es OpCodes zu Kombinationen aneinander, führt diese unter allen Anfangsbedingungen aus und wertet wieder aus, was diese Code-Kombination bewirkt hat.
Da meist längere Kombinationen zur Ziellösung notwendig sind, plant es die Codezusammen-
- 15 stellung, in dem es zielgerichtet nur die Codes zur Kombination benutzt, die keinen Schaden anrichten, also nicht den eigenen Code überschreiben und am besten überhaupt keine RAM-Schreibzugriffe machen, ferner keine verbotenen Register bzw. Operationen benutzen (ADT.unused_Registers_BitCode|ADT.unused_Operations_BitCode) und auch keine Exceptions verursachen, wo man bei Divide-Error und Overflow-Exception toleranter sein sollte. OpCodes
- 20 die gewünschte Ziel- und Quellregister benutzen, werden wiederum bevorzugt kombiniert (ADT.all_source/dest_Registers_BitCode).
ADT.aim_fulfill_valuation_mode bestimmt, ob die Bewertungsfunktion in SQL oder direkt in Maschinensprache vorliegt. Für den normalen Anwender wäre die langsamere SQL-Variante benutzerfreundlicher und der Spezialist wird für komplexere Aufgaben sicher schnelle
- 25 Zielerfüllungs-Bewertungsfunktionen in Maschinensprache bevorzugen.

1.3.7 Das dynamisch-reflexive Bewertungssystem:

- 30 **1.3.7.1 Programmierzwecknähe-Bewertung:**
Die ADT.aim_fulfilled_Flag_Function(Aim_ID), gibt TRUE zurück, wenn das Programmierzweck erreicht wurde und die VFT.Valuation_Function(Type='A', ADT.aim_fulfilled_Flag_Function, VFT.Function_ID_Chain), die einen signed-byte Wert, der besagt, wie nah die aktuelle CLT(n)-OpCode-Kombination bei den gegebenen Anfangsbedingungen der Ziellösung kommt.
- 35 Das Ergebnis wird in CLT(n).aim_valuation gespeichert und bildet im Vergleich mit der letzten CLT(n-1).aim_valuation den Gradient CLT(n).gradient_aim_valuation.
Da das Lösungsprogramm jedoch für alle Anfangsbedingungen funktionieren muß, werden die maximale und die durchschnittliche Wertvolligkeit der OpCode-Kombination als Mittelwert aller Anfangsbedingungen in CBT(n).max_aim_valuation bzw. CBT(n).avg_aim_valuation abgelegt; die
- 40 Gradienten zu den Werten der letzten CBT(n-1) bilden CBT(n).max_grad_aim_valuation und CBT(n).avg_grad_aim_valuation.
Bei jeder Bewertung wird VFT.boundary_value_counter hochgezählt, wenn ein Grenzwert von -128 bzw. +127 vergeben wurde und äquivalent low_value_counter, wenn eine Bewertung zwischen -16 und +15 auftrat.
- 45 Anhand dieser statistischen Daten und einer exakten Auswertung aller CLT(i).aim_valuation, z.B. indem man ein kleines Wertebereichs-Fenster durchlaufen läßt und die Einträge je Fensterbreite und -offset zählt, bewertet die SAC.Aim_Self_Valuation_Func nach jeder Programmierzweck-Erreichung der Bewertungs-Ergebnisse der VFT.Valuation_Function und somit deren Effizienz. Fallen z.B. die meisten Bewertungsergebnisse auf die Grenzwerte, war die
- 50 Bewertungsfunktion zu steil und sie muß abgeflacht werden, also in der VFT.Function_ID_Chain mehr Elemente mit negativem FIT.Function_Flatten enthalten. Umgekehrtes gilt, wenn die meisten Bewertungsergebnisse einen hohen VFT.low_value_counter-Wert verursachen.
Nach jeder Programmierzweckerreichung läuft somit eine Selbstbewertung der Bewertungsfunktion ab und ein weiterer Programmschritt der selbstprogrammierung der Bewertungsfunktion:
- 55 Zur Bewertungsfunktion kommen neue Elemente hinzu und manchmal muß auch eines weggelassen werden und der Steilheitsgrad wird angepaßt. Dann wird die Bewertung erneut vorgenommen und überprüft, ob die veränderte Bewertungsfunktion einen besseren

Wertebereich geliefert hätte. War der neue Wertebereich nach der Selbstbewertung durch `SAC.Aim_Self_Valuation_Function` schlechter wird die Änderung der Bewertungsfunktion verworfen und eine andere Änderung versucht. Verbesserte die Bewertungsfunktionsänderung den Wertebereich, wird die nächste Verbesserung versucht und wenn die Selbstbewertung
5 einen guten Wert ergibt, mit der nächsten Programmieraufgabe fortgefahren.

1.3.7.2 Dynamische Energiebewertungsfunktion:

Das energiespezifische Bewertungssystem ist folgendermaßen dynamisiert:

- 0.) Da die Ergebnis-Werte des Bewertungssystems hier auf den Wertebereich von `signed_byte`
10 beschränkt sind, wird die Bewertungsfunktion in eine Rahmenfunktion eingebettet:
Bewertungsergebnis := `MIN[MAX(Bewertungsfunktion, -128), +127]`
- 1.) Die Bewertungsfunktion 0-ter Ordnung ist "wie satt bin ich nach der Handlung ?":
Bewertungsfunktion(0) := `MIN[MAX(Energy_after, -128), +127]`
- 2.) Die Bewertungsfunktion 1-ter Ordnung ist "wieviel satter bin ich nach der Handlung ?":
15 Bewertungsfunktion(1) := `MIN[MAX(Energy_after - Energy_before, -128), +127]`
- 3.) Da das Energieregister vom Typ `unsigned integer` (DWord) ist, wären die Rahmen bei der Bewertung zu schnell erreicht, deshalb entweder geringer Logarithmus oder:
Bewertungsfunktion(2) := `MIN[MAX(SQRT(Energy_after - Energy_before), -128), +127]`
- 4.) Jetzt ergäben sich falsche Werte bei negativen Energie-Gradienten, deshalb 3. Wurzel oder:
20 Bewertungsfunktion(3) := `MIN[MAX(SGN(EnergyGrad) * SQRT(EnergyGrad), -128), +127]`; mit
 $EnergyGrad = Energy_after - Energy_before$
- Möglicherweise wäre auch die Funktion $\frac{1}{2} \cdot SGN(EnergyGrad) \cdot SQRT(SQRT(EnergyGrad))$ besser, da diese exakt bis zu den Rahmenwerten reicht. Aber vielleicht werden auch die Rahmenwerte fast nie erreicht und eine feinere Gliederung um den Nullwert wäre viel wichtiger.
- 25 Dies hängt davon ab, wie oft die Rahmenwerte erreicht werden und wie viele Energie-Gradienten kleine Werte aufweisen. Vielleicht muß auch der Ergebniswert stärker gewichtet werden und eine Gradientbewertung reicht allein nicht aus; ferner muß mitberücksichtigt werden, wieviele/welche weitere(n) Register neben der Energieveränderung mitbetroffen sind und welche OperationsTypen ausgeführt wurden, u.s.w., und schließlich die Ausführungszeit
- 30 der Bewertungsfunktion selbst. Deshalb muß sich das energiespezifische Bewertungssystem im Laufe der Zeit verfeinern und anpassen (wie bei intelligenten biologischen Lebensformen).
In dynamic embedded {PL/SQL ist das verändern und neu parsen der als String gespeicherten Bewertungsfunktion kein Problem. Wegen der Ausführungsgeschwindigkeit und der Implementationsfähigkeit voriger Aufgabenlösungen soll jedoch die Bewertungsfunktion zukünftig in
- 35 Maschinensprache erfolgen.
Die Programmieraufgabe der Verbesserung der energiespezifischen Bewertungsfunktion wird ebenfalls nach jeder Programmierzilerreichung angegangen.

40 Bewertungssystem und Bewertungsergebnisse sind immer reflexiv.

1.3.8 Selbstbewußtwerdung, Reproduktion und Evolution:

- 45 Durch den Selbstreperaturvorgang bei Schmerz kennt das Programm seine Lage im Speicher. Es kann nun die Auswirkungen seiner eigenen OpCodes der Reihe nach testen. Erkennt es später einmal die Gesamtauswirkung seiner ganzen Codelänge, wird es sich seiner selbst bewußt, kann seinen Code replizieren und aufgrund seines Wissens hierbei bewußt verändern (z.B. das lästige Erniedrigen des "Energie"-Registers entfernen). Die intelligente Reproduktion ist der der genetischen Reproduktion weit überlegen, da bei letzterer nur auf vorhandene DNA zurückgegriffen werden kann und hier der eigene Programmcode bewußt beliebig verändert und erweitert wird.
- 50

1.4. Programmieraufgabenstellung und Beispiele der Zielerreichung

- Dem KB-Programm wird nun eine beliebige Programmieraufgabe gestellt, und es erhält in *ADT.aim_fulfilled_Flag_Function* einen Prüfungsalgorithmus, mit dem es überprüfen kann, ob es die Aufgabe erfüllt hat.
- Seine Aufgabe ist nun, ein Programm zu schreiben, daß die gestellte Aufgabe löst.

1.4.1 Beispielaufgabe 1: Erstellung eines Programms zur Berechnung des Mittelwerts:

- 10 Eine sehr einfache, aber leicht nachvollziehbare Aufgabe für das KB-Programm könnte sein: "Schreibe ein Programm, das den Mittelwert 2er beliebiger Integer-Zahlen berechnet."
- Das KB-Programm hat diese Aufgabe erfüllt, wenn die Differenz von der Ergebnis-Zahl zur kleineren Zahl gleich der Differenz von der Ergebnis-Zahl zur größeren Zahl ist, und dies für
- 15 beliebig viele Eingabe-Zahlenpaare zutrifft.
- Das KB-Programm kennt jedoch den Befehlssatz des Prozessors nicht - ihm stehen lediglich die OpCodes zur Verfügung, die keinen Schaden verursachten und es kennt einen Teil deren Wirkungen bei einigen unterschiedlichen Anfangsbedingungen.
- Durch das Corruption-Healing oder das Energie-Register kennt es bereits einfachste Aufgaben
- 20 wie "führe keine Handlung aus, die Schmerz verursacht" oder "führe Handlungen aus, die mich satt machen".
- Zur Erreichung von wirtschaftlichen Zielen benötigt es nun Bewertungsvariablen, die ihm Dinge sagen wie
- a.) Wieviel näher oder weiter weg vom Ziel hat mich diese Befehlskombination gebracht (das
- 25 jeder einzelne Befehl davon gegenteilige Wirkung haben kann, ist hier nicht von Interesse).
- b.) Wieviele Taktzyklen habe ich für die Lösung verbraucht.
- c.) Wieviele Bytes ist mein Programm lang (wieviele OpCodes mit welchen Längen).
- Diese Tabellen-Spalten sind: *aim_valuation; cycles_of_execution; OpCode_length_or_jump*.
- 30 Die Eingabe-Werte der Beispiel-Aufgabe seien in den ersten beiden Datenregistern (EAX_{π} , EBX_{π} bzw. DO_{μ} , $D1_{\mu}$ bzw. $GPR0_{\psi}$, $GPR1_{\psi}$), i.F. $R0$ und $R1$.
- Der Ausgabe-Wert soll im dritten Datenregister (ECX_{π} | $D2_{\mu}$ | $GPR2_{\psi}$) i.F. $R2$ erfolgen. Ist die Aufgabe für beliebige Eingabewerte gelöst, ist das Programm fertig, da es sich wegen der Ein- und Ausgabevariablen um eine Funktion handelt. Bei mehreren Lösungen wird die mit den wenigsten
- 35 verbrauchten Taktzyklen gewählt.
- Die aufgabenspezifische Zielerreichungs-Bewertungs-Funktion, die zur Berechnung von *OLT.aim_valuation* benötigt wird, ist somit in diesem Beispiel:
- ADT.aim_fulfilled_Flag_Function*(Mittelwert mit o.g. Registern) = $[(R2-R0)=(R1-R2)]$
- Hier kann jedoch das Problem auftreten, daß ein Eingabe-Wert gerade und der andere ungerade
- 40 ist und die Aufgabe deshalb mit dieser Eingabe-Kombination somit manchmal keine Lösung hat.
- Das KB-Programm wird mehrere Lösungs-Programme finden und sich für dasjenige entscheiden, das am wenigsten Taktzyklen verbraucht, also das Ergebnis am schnellsten liefert.
- Möglich wäre bei CBT(3) folgendes Lösungsprogramm: *MOV R0,R2 ; ADD R1,R2 ; SHR R2*
- (natürlich im Maschinencode des verwendeten Prozessors - beim Pentium wäre das die 48-Bit-
- 45 Zahl $\$89C2.01CA.D1EA$, bei Motorola $\$2400.D282.E2C2$ und beim PowerPC eine 96-Bit Zahl)

1.4.2 Beispielaufgabe 2: Erstellung eines Programms zur Berechnung der Kubikwurzel:

- 50 Eine weitere einfache Aufgabe wäre "Schreibe ein Programm, das die Kubikwurzel aus einer reellen FFP-Zahl (32-Bit) berechnet"; die Eingabe soll in $R0$ (EAX_{π}), die Ausgabe in $R3$ (EBX_{π}) erfolgen.
- Das KB-Programm hat diese Aufgabe erfüllt, wenn die Ergebnis-Zahl mit ihrem Quadrat multipliziert den Anfangswert ergibt:
- 55 $\Rightarrow aim_fulfilled_Flag_Function(Kubikwurzel) = [(R3 * R3 * R3) = R0]$ (\leftarrow natürlich in FFP-Multipl.)
- Einen Einzelbefehl wie bei der Quadratwurzel (*FSQRT*) gibt es bei der Kubikwurzel nicht.
- Ein Ergebnis könnte bei CBT(8) beim Pentium II folgendermaßen aussehen:

Op1(16b): MOV CL,3 ;ECX=\$????:0003 [1011.0001:0000.0011]
 Op2(16b): FLD1 ;ST(0)=1.0 [1101.1001:1110.1000]
 Op3(16b): FIDIV CX ;ST(0)=1/3 [1101.1110:1111.0001]
 Op4(16b): FLD EAX ;ST(0)=R0 ;ST(1)=1/3 [1101.1001:1100.0000]
 5 Op5(16b): FYL2X ;ST(0)=1/3 log₂(R0) [1101.1001:1111.0001]
 Op6(16b): FLD1 ;ST(0)=1.0 ;ST(1)=1/3 log₂(R0) [1101.1001:1110.1000]
 Op7(16b): FSCALE ;ST(0)=1.0*2^[1/3 log₂(R0)] [1101.1001:1111.1101]
 Op8(16b): FST EBX ;EBX=2^[1/3 log₂(R0)] [1101.1001:1101.1011]
 (natürlich nur die 2.Spalte als 128-Bit-Zahl.)
 10 Hexadezimal wäre das B103.D9E8.DEF1.D9C0:D9F1.D9E8.D9FD.D9DB.
 Dies wäre eine mögliche Lösungszahl (=Programm) für die gestellte Aufgabe (es gibt bestimmt auch kürzere oder schnellere Lösungen).

1.5. Festplatten-Speicherbedarf und Vergessen

In beiden Beispielen wäre man zwar mit 16-Bit-Befehlen ausgekommen, jedoch wird ersichtlich, daß bei größeren Programmieraufgaben der Festplattenspeicher knapp wird. Deshalb wird das KB-Programm unwichtige OpCode-Kombinationen vergessen müssen.

1.5.1 Tabellengrößen:

IST, RIT und CIT fallen kaum ins Gewicht.

Theoretisch könnte $size(OBT) = 2^{32} \cdot \sum \text{Bytes}(\text{Spalte}(i)) = 485 \text{ GB}$ groß werden, jedoch sind auch bei einem RISC-Prozessor nie alle 32-Bit-Kombinationen als Befehl genutzt und realistisch sind im Mittel bei RISC-Prozessoren c.a. 28 Bit \Rightarrow 30 GB und bei CISC-Prozessoren 20 Bit \Rightarrow 118 MB (die meisten sind dort 16 Bit-Befehle; es gibt wenige 8-Bit- und einige 24- und 32-Bit-Befehle, und längere als 32-Bit-Befehle werden hier nicht berücksichtigt).

Bei den 62 Anfangsbedingungen kann $size(OLT) = 2^{(20..28)} \cdot 62 \cdot \sum \text{Bytes}(\text{Spalte}(i)) = 3$ (CISC) .. 832 (RISC) GB groß werden und $size(ORT)$ c.a. genauso groß, wenn man bedenkt, daß durch ein OpCode meist ein Zielregister und ein Quellregister betroffen ist, möglicherweise auch keines oder nur ein Zielregister und selten mehrere Register. Da es jedoch mehrere mögliche zugehörige *Operation_BitCodes* geben könnte, würde sich die Tabellengröße erhöhen, wenn dies nicht durch die vielen OpCodes, die eine Exception auslösen, kompensiert würde.

Ein weitaus größeres Problem könnte das exponentielle Wachstum der $CxT(i)$ darstellen, da für jedes i ein Faktor von $2^{(20..28)}$ hinzukommt. Dies wird jedoch durch das Bewertungssystem kompensiert, das mit abnehmendem Restspeicher seine Toleranz einschränken kann - so können Kombinationen von vornherein ausgeschlossen werden, die Codes bzw. Kombinationen mit geringer $CBT(i).max_aim_valuation$ (bzw. $avg_aim_valuation$) kombinierten würden.

Auch wenn der Speicherbedarf jetzt noch hoch erscheinen mag, dürfte dies in naher Zukunft kein Problem mehr darstellen. Auch die mit den Tabellengrößen und Kombinationsmöglichkeiten wachsenden Rechenzeiten werden durch immer größer und schneller werdende Festplatten und immer leistungsfähigere Rechner bewältigbar.

1.5.2 Vergessen:

Wie bei allen intelligenten Lebensformen muß das System unwichtige und weniger wichtige Informationen vergessen können, weil

- a.) der Speicherplatz begrenzt ist und
- b.) die Zugriffszeiten werden unnötig langsam.

Deshalb werden nach jeder zufriedenstellenden Zielerreichung, bei Eingabe eines neuen Ziels, die ELT und alle $CxT(i)$ -Tabellen ab einem restspeicherabhängigen Grad gelöscht und die Codekombinationen bzgl. der neuen Programmieraufgabe in den verbleibenden $CxT(i)$ neu

bewertet und ab der gelöschten CxT inkrementell dynamisch wieder neu angelegt.

1.6. Bewußtwerdung

Durch `try_and_error` lernt das Programm was jede einzelne Handlung bewirkt und was Handlungsfolgen bewirken.

- 5 Im Rahmen des Corruption-Healings (bei versehentlichem Eigencode-Überschreiben) muß es seinen Code reparieren und kennt so seine Position im Speicher. Wenn es einmal die Wirkung der Handlungsfolge seines eigenen Codes erkennt, wird es seiner selbst bewußt und kann seinen Code reproduzieren und bei der Reproduktion bewußt verbessern.
- 10 Durch die so initiierte Evolution entsteht eine immer komplexere Form des künstlichen Bewußtseins, das immer größere Aufgaben bewältigen kann.

1.7. Darstellung der wirtschaftlichen Vorteile

- 15 Es handelt sich hier um ein vollkommen neues Feld der Computer-Verwendung. Während im Computer normalerweise von Menschen programmierte Programme ablaufen, die anwendergesteuerte Anweisungen ausführen, programmiert das KB-Programm selbst programmierzielorientiert Handlungen, die zukünftig ausgeführt werden sollen.
- 20 Der Bedarf an Softwareentwicklung ist weltweit viel größer als das menschliche Potential an Software-Entwicklern.
Ein Programm, das lernt, Programme selbst zu schreiben (und das gleich in Maschinencode), kann kleine Programmier-Aufgaben selbst lösen und wird im Laufe seiner "Evolution", wenn man ihm ausreichend Speicherplatz läßt, auch selbständig lernen, komplexe Programme zur
- 25 Lösung großer Programmieraufgaben zu generieren.

2. Patentansprüche:

1. Ein Computer-Programm, das selbsttätig Maschinencode erstellt und ausführt, dadurch gekennzeichnet, daß
5 es den auszuführenden Maschinencode nicht aus einem Quellcode übersetzt und auch nicht als Daten einliest und es den Maschinencode auch nicht aufgrund einer gegebenen Logik des Programmierers erstellt, sondern daß es Zahlen generiert und diese als Maschinencode ausführt und die Wirkung der Maschinencode-Ausführung analysiert.
2. Ein System nach Anspruch 1, dadurch gekennzeichnet, daß das System mehrere solcher
10 Zahlenkombinationen als Maschinencodebefehle aneinanderreihet, und die Auswirkung der Codekombination analysiert.
3. Ein System nach Anspruch 2, dadurch gekennzeichnet, daß das System mehrere solcher Maschinencodekombinationen aneinanderreihet, und die Auswirkungen der miteinander kombinierten Codekombinationen analysiert.
- 15 4. Ein System nach Anspruch 1 bis 3, dadurch gekennzeichnet, daß dem System eine Programmieraufgabe gestellt wird, und es bewertet, wie nah die analysierte Zahlencodekombination der Lösung der gestellten Programmieraufgabe kommt.
5. Ein System nach Anspruch 1 bis 4, dadurch gekennzeichnet, daß das System an diese Zahlencodes oder Codekombinationen gezielt diejenigen analysierten Zahlencodes oder Code-
20 kombinationen anfügt, die aufgrund der analysierten Operationsart, den analysierten Quell- und Zielregistern und der ermittelten Bewertung aus Anspruch 4, anfügt, die aufgrund diesen Werten eine hohe Wahrscheinlichkeit haben, daß die Gesamtkombination die gestellte Aufgabe an ehesten löst.
6. Ein System nach Anspruch 5 das wiederum die Wirkung der Gesamtkombination analysiert und bzgl. der Zielerreichungsnähe bewertet.
7. Ein System nach Anspruch 1, in dem Grundbedürfnisse äquivalent "kein Schmerz" (=keine Beschädigung des Systems) und "kein Hunger" (=kein drohender Energieverlust) folgendermaßen modelliert sind:
 - a. Schmerz, modelliert durch das überschreiben, und dadurch wiederum notwendigwerdende
30 reparieren, des eigenen Programmcodes, was Zeit kostet und das unter b. modellierte "Energie"-Register schneller dekrementiert
 - b. Hunger, modelliert durch die stetige zeitabhängige Abnahme eines Registerwerts und negativen Systemauswirkungen bei niedrigen Werten, wie
 - den Verlust der Fähigkeit der Bewertung von Zahlencodekombinationen bzgl. der Zielerreichung bei niedrigen Werten,
 - 35 - Fehler bei der Bewertung der betroffenen Quell- und Zielregister, sowie der Operationsart bei sehr niedrigen Werten,
 - den Verlust der Fähigkeit der Selbstreperatur (bei "Schmerz" - siehe a.) bei extrem niedrigen Werten,
 - 40 - Abnahme der Spannungsversorgung des RAM durch hardwaremäßig an dieses Register gekoppelten variablen Widerstand, der sich bei zwei mal in Folge auftretenden extrem niedrigen Werten im "Energie"-Register erhöht.
 - Abnahme der Spannungsversorgung des Prozessors durch hardwaremäßig an dieses Register gekoppelten variablen Widerstand, der sich bei drei mal in Folge auftretenden
45 extrem niedrigen Werten im "Energie"-Register erhöht.
8. Ein System nach den Ansprüchen 1 bis 7, mit dem Unterschied, daß dem System keine Programmieraufgabe gestellt wurde, und die Zielerreichungs-Bewertung nun bei Maschinencodes und Codekombinationen positiv ausfällt, die keinen "Schmerz" verursachen und das
50 "Energie"-Register erhöhen, und Kombinationen um so negativer bewertet, je mehr eigen- genutzten Speicher sie überschreiben und je mehr sie das "Energie"-Register erniedrigen.
9. Ein System nach Anspruch 8, das nicht nur wie oben Code generiert, sondern auch bestehenden vorgegebenen Code analysiert, wie z.B. seinen eigenen, um zu bewerten, was seine Codeabschnitte und auch sein Gesamtcode bewirkt.
10. Ein System nach o.g. Ansprüchen, bei dem die Bewertungsfunktionen bzgl. der Systemziele
55 (Programmierziel, Energiespezifische Handlung, u.s.w.) dynamisch reflexiv sind, also neben der Verfolgung der Erreichung der Programmierziele und positiver energiespezifischer Handlungen (und ggf. weiterer Aspekte) Selbstbewertungsroutinen durchgeführt werden, die

die Bewertungsfunktionen anhand ihrer Bewertungsergebnisse bewerten und die Bewertungsfunktionen verändern, testen und wieder bewerten und dann eine verbesserte Bewertungsfunktion auswählen, um bei der nächsten Programmieraufgabe effizienter zu sein.

- 5 11. Ein System nach Anspruch 10, bei dem das Bewertungssystem selbst Programmieraufgaben stellen kann, deren Ergebnisroutine als Teilfunktionen der Bewertungsfunktion dienen kann, um die Bewertungsfunktion selbst zu verbessern.
- 10 12. Ein System nach Ansprüchen 1 bis 11, bei dem zusätzlich über eine Tabelle der Betriebssystemroutinen die Funktionen, Ein- und AusgabeRegister und Einsprungsadressen der System-Routinen zur Verfügung stehen und so als CALLS vom KB-Programm in den Lösungscode implementiert werden können.
- 13. Ein System nach o.g. Ansprüchen, in dem mehrere solcher Programme parallel laufen und das Erlernte aus den Opcode- und Kombinations-Ausführungen austauschen können.
- 14. Ein System nach Anspruch 13, in dem mehrere Rechner, auf denen je eins oder mehrere
15 solcher Programme laufen, miteinander vernetzt sind.
- 15. Ein System nach Anspruch 8, 11, 12, 13 oder 14 in dem wieder ein Programmierzil vorgegeben ist, deren Erreichung jedoch nicht wie in Anspruch 5 oder 6 durch eine Zielannäherungsbewertung bewerkstelligt wird, sondern in dem bei Codekombinationen, die sich vom Programmierzil entfernen, eine Erniedrigung des Energieregisters verursachen und
20 Codekombinationen, die in Richtung der Erreichung des vorgegebenen Programmierzils gehen eine Erhöhung des Energieregisters mit sich bringen.

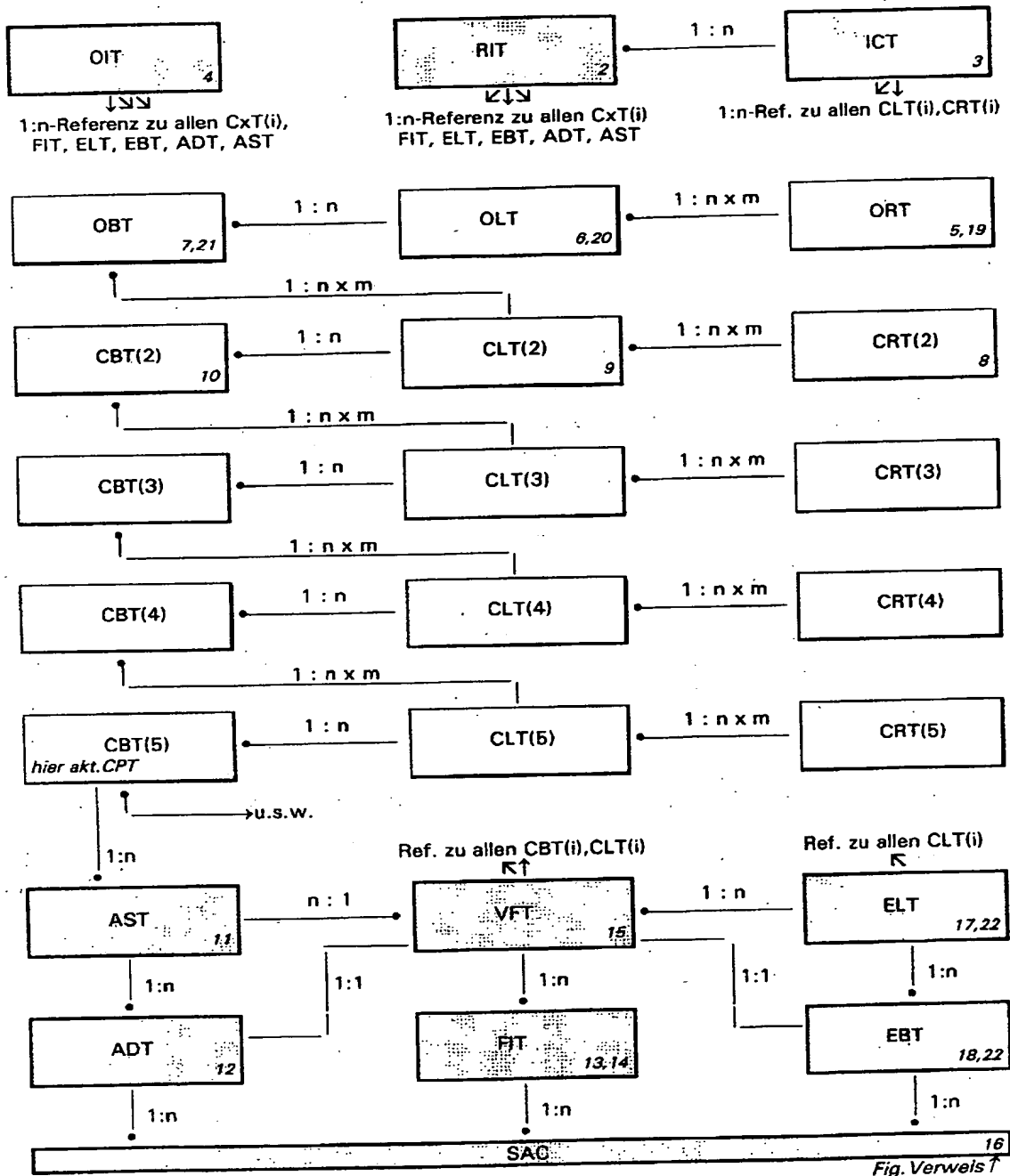
3. Zeichnungen:**3.1 Relationale Datenbank des KB-Wissens:****3.1.1 ER-Diagramm der KB-Datenbank:**

Fig. 1

Fig. Verweis ↑

3.1.2 Tabellen der KB-Datenbank:

Register-Identifikations-Tabelle: [RIT: jedem Prozessor-Reg. wird eine Reg.ID + Reg.BitCode zugeordnet]

| Spalte: | Datentyp | Wertebereich | Bedeutung: |
|----------------------|--------------|------------------------|--|
| Register_ID (PK) | signed byte | -128..127 | 0 = Flags-Reg., 1-... = Daten-Reg., Adr.Reg., Adr.Reg.-Verweise, FFP-Reg., Control-Reg., Debug-Reg., u.s.w.; neg.Reg.ID = Exc.Vect.Nr. (kein Reg.) |
| Register BitCode | number | 0..2 ¹²⁸ -1 | 2*Register ID, 0 bei neg. Register ID. |
| Register_type | char(1) | 1 Byte | Typ des Registers: # = Flags-Reg., D = Daten-Reg., A = Adress-Reg., V = Adr.-Reg.-Verweis, E = Exception-Vector, u.s.w. (o.ä., je nach Prozessor). |
| Register number | byte | 0..127 | Nummer des (Register-Type)-Registers. |
| Register Description | varchar2(32) | ≤32 Bytes | optionale Beschreibung des Register[-Verweise]s. |

Fig.2a

| Register ID | Register BitCode | Register type | Register number | Register Description |
|-------------|------------------|---------------|-----------------|--|
| ... | 0 | E | ... | {for all Exception-Vectors} |
| -8 | 0 | E | 8 | Privilege-Violation Exception |
| ... | 0 | E | ... | {for all Exception-Vectors} |
| 0 | 1 | # | 0 | Status-Register (Δ EFlags _x) |
| 1 | 2 | D | 0 | Data-Register D0 |
| ... | 4 | D | ... | {for all Data-Registers D1-D6} |
| 8 | 8 | D | 7 | Data-Register D7 |
| 9 | 16 | A | 0 | Adress-Register A0 |
| ... | ... | A | ... | {for all Adress-Registers A1-A6} |
| 16 | 65536 | A | 7 | Adress-Register A7 (= USP) |
| 17 | 131072 | V | 0 | Destination of Adress-Register A0 |
| ... | ... | V | ... | {for all Adr.-Reg.-Destinations A1-A6} |
| 24 | 16777216 | V | 7 | Destination of Adr.-Reg. A7 [= (USP)] |
| 25 | 33554432 | v | 0 | Destination before Adr.Reg A0 [- (A0)] |
| ... | ... | v | ... | {for all Adr.Reg.-Dest. before A1-A6} |
| 32 | \$1.0000.0000 | v | 7 | Destination before Adr.Reg A7 [- (USP)] |
| 33 | \$2.0000.0000 | F | 0 | Floating-Point Data-Register FPR0 |
| ... | ... | ... | ... | {for all further Registers} |

Fig.2b (RIT am Beispiel Motorola)

Abfangsbedingungen-Tabelle: [ICT: jedem Register werden 62 Anfangsbed. (initial condit.) zugeordnet]

| Spalte: | Datentyp | Wertebereich | Bedeutung: |
|------------------|-------------|-----------------------|--|
| IniConNr (PK) | signed byte | -31..+30 | Anfangsbedingungen-Nr. |
| Register ID (PK) | signed byte | 0..127 | siehe RIT |
| Register Value | integer | 0..2 ³² -1 | Wert des Registers bei dieser Anfangsbedingungen-Nr. |

Fig.3a

| | | | |
|--|--|--|--|
| Dafür werden 62*RegisterAnzahl Testwerte generiert, z.B. mittels folgender Funktion: | | | |
| Register_Value(IniConNr, Register_ID) = SGN(IniConNr) * INT(2*ABS(IniConNr/2) + ½) + Primzahl(3*Register_ID) | | | |
| , mit Primzahl(0)=0 und Primzahl(-n) = -Primzahl(n); keine 2 gleichen Register[-Verweis]-Inhalte. | | | |

Fig.3b

Operations-Identifikations-Tab.: [OIT: je Prozessor-Oper. wird eine Oper.ID + Oper.BitCode zugeordnet]

Spalte: Datentyp Wertebereich Bedeutung:

| Operation ID (PK) | Datentyp | Wertebereich | Bedeutung |
|------------------------|--------------|--------------|--|
| Operation BitCode (FK) | signed byte | -1..63 | Bit des Calculation BitCode - siehe Tabellendaten. |
| Operation Type | number | 0..2128 | 2 ⁿ Calculation ID - siehe Tabellendaten. |
| Operation Mnemonic | char(5) | 5 Bytes | 5-Character-Code des Operations-Typs, siehe Fig.4c |
| Operation Description | char(5) | 5 Bytes | Abkürzung der Rechenoperation - siehe Tab.Daten. |
| | varchar2(32) | ≤32 Bytes | optionale Beschreibung der Rechenoperation (s.u.). |

Fig.4a

| Operation ID | Operation BitCode | Operation Type | Op.Mnemonic | Operation Description |
|--------------|-------------------|----------------|-------------|--|
| -1 | 0 | | ??? | unbekannte Operation |
| 0 | 1 | :I11? | TST | Flags in Reg.[Verw.] - Abhängigk.setz. |
| 1 | 2 | :I12! | NEG | Negation Betragsbildung |
| 2 | 4 | :I12! | NOT | bitweise Invertierung |
| 3 | 8 | :I02 | MOVI | feste Zahl → Register(verweis) |
| 4 | 16 | :I12+ | ADDI | feste Zahl addieren |
| 5 | 32 | :I12- | SUBI | feste Zahl subtrahieren |
| 6 | 64 | :I13* | MULI | mit fester Zahl multiplizieren |
| 7 | 128 | :I23/ | DIVI | durch feste Zahl dividieren |
| 8 | 256 | :I13% | MODI | Divisionsrest einer festen Zahl |
| 9 | 512 | :I12* | SHLI | FestZahl-mal verdoppeln |
| 10 | 1.024 | :I12/ | SHRI | FestZahl-mal halbieren |
| 11 | 2.048 | :I12 | ORI | Bits einer festen Zahl hinzufügen |
| 12 | 4.096 | :I12& | ANDI | Bits einer festen Zahl löschen |
| 13 | 8.192 | :I12? | BTSTI | Reg.[Verw.] - Vergleich m. festem Bit |
| 14 | 16.384 | :I12? | CMPI | Reg.[Verw.] - Vergleich m. fester Zahl |
| 15 | 32.768 | II22 | MOV | Quell-Reg.[V.] → Ziel-Reg.[V.] |
| 16 | 65.536 | II22+ | ADD | Reg.[Verw.] - Addition |
| 17 | 131.072 | II22- | SUB | Reg.[Verw.] - Subtraktion |
| 18 | 262.144 | II23* | MUL | Reg.[Verw.] - Multiplikation |
| 19 | 524.288 | II33/ | DIV | Reg.[Verw.] - Division |
| 20 | 1.048.576 | II33% | MOD | Reg.[Verw.] - Divisionsrest |
| 21 | 2.097.152 | II22* | SHL | Reg.[Verw.] - mal verdoppeln |
| 22 | 4.194.304 | II22/ | SHR | Reg.[Verw.] - mal halbieren |
| 23 | 8.388.608 | II22 | OR | Reg.[Verw.] - Bits hinzufügen |
| 24 | 16.777.216 | II22& | AND | Reg.[Verw.] - Bits löschen |
| 25 | 33.554.432 | II21? | BTST | Reg.[Verw.] - Vergleich m. Reg.-Bit |
| 26 | 67.108.864 | II21? | CMP | Reg.[Verw.] - Vergleich m. Reg.[Verw.] |
| 27 | 134.217.728 | :P00. | JMP | addiere feste Zahl → PC _n /EIP _n |
| 28 | 268.435.456 | CP1.< | JLT | Jump if CMP < |
| 29 | 536.870.912 | CP1!> | JLE | Jump if CMP ≤ |
| 30 | 1.073.741.824 | CP1.= | JEQ | Jump if CMP = |
| 31 | 2.147.483.648 | CP1!< | JGE | Jump if CMP ≥ |
| 32 | 4.294.967.296 | CP1!= | JNE | Jump if CMP ≠ |
| 33 | \$2.0000.0000 | CP1.> | JGT | Jump if CMP > |
| 34 | \$4.0000.0000 | CP1!< | JPL | Jump if ≥ 0 |
| 35 | \$8.0000.0000 | CP1.< | JMI | Jump if < 0 |
| 36 | \$10.0000.0000 | CP1.^ | JCS | Jump if Carry set |
| 37 | \$20.0000.0000 | CP1!^ | JCC | Jump if Carry clear |
| 38 | \$40.0000.0000 | CP1.~ | JVS | Jump if Overflow set |
| 39 | \$80.0000.0000 | CP1!~ | JVC | Jump if Overflow clear |
| 40 | \$100.0000.0000 | CP2.< | DJMP | Decrement and Jump if Reg. < 0 |
| 41 | \$200.0000.0000 | SP1.. | CALL | PC _n /EIP _n → -(USP _n /ESP _n); + JUMP |
| 42 | \$400.0000.0000 | SP11. | RET | (USP _n /ESP _n) + → PC _n /EIP _n |
| 43 | \$800.0000.0000 | .I... | I??? | unbek. Integer-Operation |
| 44 | \$1000.0000.0000 | .F... | F??? | unbek. Floating-Point-Operation |

| | | | | |
|-----|-----------------------------|-------|-------|--|
| 45 | \$2000.0000.0000 | FF09 | FINIT | init FloatingPoint-Unit |
| 46 | \$4000.0000.0000 | FI12 | FIST | store Float.Point-Reg. → Integer-Reg. |
| 47 | \$8000.0000.0000 | IF12 | FILD | load Integer-Reg. → FloatingPoint-Reg. |
| 48 | \$1.0000*2 ³² | IF22+ | FIADD | FloatingPoint add Integer |
| 49 | \$2.0000*2 ³² | IF22- | FISUB | FloatingPoint sub Integer |
| 50 | \$4.0000*2 ³² | IF22* | FIMUL | FloatingPoint multipl. mit Integer |
| 51 | \$8.0000*2 ³² | IF22/ | FIDIV | FloatingPoint teile durch Integer |
| 52 | \$10.0000*2 ³² | IF21? | FICMP | Float.Point-Compare Integer → Flags |
| 53 | \$20.0000*2 ³² | :F02 | FLD# | Konstante → FloatingPoint-Register |
| 54 | \$40.0000*2 ³² | .F12! | FABS | FloatingPoint-Betragsbildung |
| 55 | \$80.0000*2 ³² | FF12 | FLD | FloatingPoint-Kopieren |
| 56 | \$100.0000*2 ³² | FF22+ | FADD | FloatingPoint-Addition |
| 57 | \$200.0000*2 ³² | FF22- | FSUB | FloatingPoint-Subtraktion |
| 58 | \$400.0000*2 ³² | FF22* | FMUL | FloatingPoint-Multiplikation |
| 59 | \$800.0000*2 ³² | FF22/ | FDIV | FloatingPoint-Division |
| 60 | \$1000.0000*2 ³² | .F12@ | FSQRT | FloatingPoint-Wurzel |
| 61 | \$2000.0000*2 ³² | .F12@ | FSIN | FloatingPoint-Sinus |
| 62 | \$4000.0000*2 ³² | .F12@ | FCOS | FloatingPoint-Cosinus |
| 63 | \$8000.0000*2 ³² | .F12@ | FATAN | FloatingPoint-ArcusTangens |
| 64 | \$1*2 ⁴⁸ | FF22* | FEXP2 | y: = y*2 ^x , o.ä. Exponentialfunktion |
| 65 | \$2*2 ⁴⁸ | FF22/ | FLOG2 | y: = x*log ₂ y, o.ä. Logarithmusfkt. |
| 66 | \$4*2 ⁴⁸ | FF21? | FCMP | FloatingPoint-Compare → Flags |
| 67 | \$8*2 ⁴⁸ | \$I11 | SMOV | Move from a special Register |
| 68 | \$10*2 ⁴⁸ | \$I11 | MOVS | Move to a special Register |
| ... | ... | ... | ... | ... |

Fig. 4b

... mit dem Operation-Type Character-Code:

| | |
|---|---|
| 1. Zeichen = Quelle, 2. Zeichen = Ziel, mit: | <p>? = unbekannt, Platzhalter für alle möglichen folgenden</p> <p>. = nichts</p> <p>:</p> <p>I = Integer-Register-[Verweis]-Inhalt</p> <p>F = Floating-Point Register</p> <p>C = ConditionCode-Register (unterstes Byte v. EFlags₇/SR₁₀)</p> <p>P = InstructionPointer/Programmzähler (EIP₇/PC₁₀)</p> <p>S = StackPointer-Verweis</p> <p>~ = Vergleichsoperation → Flags</p> <p>\$ = ein Spezial-Reg., wie Flags-, Control-, Debug-, ...</p> <p>! = <i>Nicht</i> für Vergleichsabfrage im 4. Feld</p> |
| 3. Zeichen = Anzahl der betr. Quell-Reg. | icl. des Zielregisters, wenn auch als Source verwendet. |
| 4. Zeichen = Anzahl der betr. Ziel-Reg. | mit Flags-Register ohne Instruction-Pointer. |
| 5. Zeichen = Rechen-Wirkung: | <p>? = unbekannt</p> <p>. = keine</p> <p>! = Betragsbildung Negation bitweise Invertierung</p> <p>+</p> <p>-</p> <p>*</p> <p>/</p> <p>% = Divisionsrest</p> <p> = Bits setzen</p> <p>& = Bits löschen</p> <p>@ = trigonometrische- oder Potentialfunktion</p> <p>> = größer ? (Abfrage der Flags)</p> <p>< = kleiner ? (Abfrage der Flags)</p> <p>= = gleich ? (Abfrage der Flags)</p> <p>^ = Carry ? (Abfrage der Flags)</p> <p>~ = Overflow ? (Abfrage der Flags)</p> |

Fig. 4c

OpCode-Register-Tabelle: [ORT - von diesem OpCode + Anfangsbed. betroffene Register + Wirkung]

| Spalte: | Datentyp | Wertebereich | Bedeutung: |
|-------------------------|-------------|------------------------|--|
| OpCode (PK) | integer | 0..2 ³² -1 | Complete Instruction, truncated if > 4 Bytes |
| IniConNr (PK) | signed byte | -31..30 | Anfangsbedingungen-Nr. |
| Register ID dest (PK) | signed byte | 0..127 | ein betroffenes Ziel-Register der Ausführung, s. RIT. |
| Register ID source (PK) | signed byte | -1,0..127 | -1 oder ein mögliches Quell-Register, siehe RIT. |
| value before change | integer | 0..2 ³² -1 | Wert von Geändertem vor Änderung. |
| value after change | integer | 0..2 ³² -1 | Wert von Geändertem nach Änderung. |
| gradient if unsigned | signed byte | -128..127 | Erhöhungs-Gradient, wenn als unsigned definiert. |
| gradient if signed | signed byte | -128..127 | Erhöhungs-Gradient, wenn als signed definiert. |
| value_source | integer | 0..2 ³² -1 | Wert von möglichem Quell-Register-[Verweis]-Inhalt |
| Operations_BitCode | number | 0..2 ¹²⁸ -1 | Bitmaske, die alle zutreffenden Rechenarten dieser Register_ID_dest / Register ID source -Kombination kennzeichnet (z.B.: 2 + 2 = 2 * 2 bei gleichen Reg's). Werte siehe CIT, Berechnung siehe Fig.19. |

Fig.5

Für jede Register- oder Register-Verweisinhalts-Änderung derselben Ausführung gibt es hier einen Eintrag, der die Register-[Verweis]-Werte vor und nach der Ausführung, sowie dessen Änderungsgrad angibt, und einen Hinweis ein mögliches Quellregister und auf die betreffende Operation, die stattgefunden haben könnte. (Packed, Teil-Word/Byte und BCD werden nicht berücksichtigt.)

OpCode-Lern-Tabelle: [OLT - ermittelte Wirkung des OpCodes bei diesen Anfangsbed.]

| Spalte: | Datentyp | Wertebereich | Bedeutung: |
|--------------------------|-------------|------------------------|---|
| OpCode (PK) | integer | 0..2 ³² -1 | Complete Instruction, truncated if > 4 Bytes |
| IniConNr (PK) | signed byte | -31..30 | Anfangsbedingungen-Nr. |
| active ChkSum corrupt | boolean | 1 0 | Flag: aktives KB-Progr. CheckSum changed |
| inactive ChkSum corrupt | boolean | 1 0 | Flag: inaktives KB-Progr. CheckSum changed |
| Exception Vect changed | signed byte | -128...0 | Register ID d. 1. überschriebenen Exception-Vectors |
| multiple Exc Vect chg. | boolean | 1 0 | min.2 Exception-Vektoren wurden überschrieben. |
| Processor Mode Changed | boolean | 1 0 | Flag: Prozessor-Modus wurde verändert (z.B. Trace) |
| Number of Exception | byte | 0..N+1 | Exception-Nummer(ggf. je + 1) [0: = keine Exc.] |
| OpCode_length_or_jump | signed byte | -128..127 | EIP _{PCu} jetzt N Bytes weiter bzw. zurück; -128 = \$FF = langer Sprung zurück; 127 = \$7F = langer vor. |
| CCR before execution | byte | 0..255 | Flags, die einen Sprung ausgelöst haben könnten. |
| Register_changed_BitCode | number | 0..2 ¹²⁸ -1 | $\exists 2^{\text{ORT.Register ID dest}} \vee \text{ORT(OpCode,LfdNr)}$ |
| Register_source_BitCode | number | 0..2 ¹²⁸ -1 | $\exists 2^{\text{ORT.Register ID source}} \vee \text{ORT(OpCode,LfdNr)}$ |
| max_Operations_BitCode | number(19) | 0..2 ¹²⁸ -1 | $\exists \text{ORT.Calculation BitCode} \vee \text{ORT(OpCode,IniConNr)}$ |
| min_Operations_BitCode | number(19) | 0..2 ¹²⁸ -1 | $\zeta \text{ORT.Calculation BitCode} \vee \text{ORT(OpCode,IniConNr)}$ |
| time of execution | integer | 0..2 ³² -1 | DeciSeconds after (20.9.1994, 0:00:00,0 Uhr) |
| cycles of execution | byte | 1..255 | Taktzyklen der OpCode-Ausführung |
| aim_valuation | signed byte | -128..127 | Ziel-Erreichungs-Bewertung bei diesen Anfangsbed. |
| gradient aim valuation | signed byte | -128..127 | Unterschied zu $CLT(n-1, IniConNr).aim_valuation$ |

Fig.6

Das letzte Adress-Reg. ist der StackPointer. Das letzte Daten-Register sei das "Energie"-Register. Als Adress-Register sei hier jedes Register bezeichnet, dessen Inhalt nicht nur ein Wert, sondern auch eine Adresse im Speicher sein kann, auf dessen Inhalt zugegriffen werden kann.

Es können mehrere Register gleichzeitig verändert worden sein, deshalb zusätzliche 1:n-Tabelle, bei der Register_ID_dest die Identität des geänderten Registers darstellt. Als mögliches Quellregister für die Änderung können u.U. mehrere in Frage kommen - diese Menge erhöht sich weiter durch die Summe aller möglichen Operationen.

OpCode-Basis-Tabelle: [OBT - ermittelte Wirkung der OpCode-Ausführung aus versch. Anfangsbed.]

Spalte: Datentyp Wertebereich Bedeutung:

| | | | |
|----------------------------|---------------|------------------------|---|
| OpCode | (PK): integer | 0..2 ³² -1 | Complete Instruction, truncated if > 4 Bytes |
| Execution counter | byte | 0..255 | Anzahl der OLT-Einträge bis jetzt |
| FatalError_counter | byte | 0..255 | Anzahl der verursachten schweren Fehler: Checksum_corrupt, Exception_Vect_changed, Trace_Bit_cheared, Processor_Mode_changed, Exceptions außer Divide-Error, Overflow. |
| low Error counter | byte | 0..255 | Anzahl der Divide-Error oder Overflow -Exceptions |
| Jump longOp probability | signed byte | -128..127 | Wahrscheinlichkeit: Befehl ist langer Opcode Sprung |
| avg OpCpde jump length | signed byte | -128..127 | mittlere OpCode/Sprung-Länge von allen Ausführungen |
| OpCode len unconfirmed | boolean | 1 0 | min. eine Abweichung von der OpCode-Länge |
| avg cycles of execution | byte | 1..255 | mittlerer Zeitverbrauch in Taktzyklen |
| exec cycles unconfirmed | boolean | 1..0 | min. eine Abweichung von der Anzahl der Taktzyklen |
| Register write probability | signed byte | -128..127 | Wahrscheinlichkeit: Befehl schreibt in Register |
| Register copy probability | signed byte | -128..127 | Wahrscheinlichkeit: Befehl kopiert Register |
| Memory write probability | signed byte | -128..127 | Wahrscheinlichkeit: Befehl schreibt in Speicher |
| Memory copy probability | signed byte | -128..127 | Wahrscheinlichkeit: Befehl kopiert Speicher |
| Reg to Mem probability | signed byte | -128..127 | Wahrscheinlichkeit: Befehl kopiert Register d.Adr.Reg. |
| Mem to Reg probability | signed byte | -128..127 | Wahrscheinlichkeit: Befehl kopiert d.Adr.Reg. in Reg. |
| Multi Reg write prob | signed byte | -128..127 | Wahrsch.: Befehl schreibt in mehrere Register. |
| Multi Mem write prob | signed byte | -128..127 | Wahrsch.: Befehl schreibt durch mehrere Adr.Reg. |
| Multi Reg to Mem prob | signed byte | -128..127 | Wahrsch.: Befehl kopiert min.2 Reg. d.min.2 Adr.Reg. |
| Multi Mem to Reg prob | signed byte | -128..127 | Wahrsch.: Bef. kopiert d.min.2 Adr.Reg. in min.2 Reg. |
| all_Reg_dest_BitCode | number | 0..2 ¹²⁸ -1 | Σ OLT.Register changed Bitcode ∨ OLT(OpCode) |
| cut_Reg_dest_BitCode | number | 0..2 ¹²⁸ -1 | ζ OLT.Register changed Bitcode ∨ OLT(OpCode) |
| all_Reg_source_BitCode | number | 0..2 ¹²⁸ -1 | Σ OLT.Register source Bitcode ∨ OLT(OpCode) |
| cut_Reg_source_BitCode | number | 0..2 ¹²⁸ -1 | ζ OLT.Register source Bitcode ∨ OLT(OpCode) |
| max Operation BitCode | number | 0..2 ¹²⁸ -1 | Σ OLT.max Operation BitCode ∨ OLT(OpCode) |
| min Operation BitCode | number | 0..2 ¹²⁸ -1 | ζ OLT.min Operation BitCode ∨ OLT(OpCode) |
| all Operation BitCode | number | 0..2 ¹²⁸ -1 | Σ OLT.min Operation BitCode ∨ OLT(OpCode) |
| cut Operation BitCode | number | 0..2 ¹²⁸ -1 | ζ OLT.max Operation BitCode ∨ OLT(OpCode) |
| max write value | integer | 0..2 ³² -1 | Maximum aller geschriebenen Werte |
| min write value | integer | 0..2 ³² -1 | Minimum aller geschriebenen Werte |
| avg write value | integer | 0..2 ³² -1 | Mittelwert aller geschriebenen Werte |
| max write gradient | integer | 0..2 ³² -1 | maximale Differenz des geänderten Werts |
| min write gradient | integer | 0..2 ³² -1 | minimale Differenz des geänderten Werts |
| avg write gradient | integer | 0..2 ³² -1 | durchschnittliche Differenz des geänderten Werts |
| evaluated source Register | signed byte | -1,0..127 | Quellregister ID (nach OBT-Auswertung) |
| evaluated_source_NumReg | signed byte | -128, -1, 0..127 | -128 ≙ LOB ist feste Quellzahl; 0..127 = weitere Quellregister ID; -1 = -1 (nach OBT-Auswertung) |
| evaluated dest Register | signed byte | -1, 0..127 | Zielregister nach OBT-Auswertung |
| evaluated_dest_Register2 | signed byte | -1, 0..127 | mögl. 2.Zielregister nach OBT-Auswertung oder Flags (bei min. 2 echten ZielReg. wird Flags nicht genannt). |
| evaluated Operation ID | signed byte | -1,0..63 | wahrscheinlichste ausgeführte Operation (Ausw.) |
| Confirmation counter | byte | 0..255 | gleiche Auswirkung bei neuen Anfangsbedingungen |
| max aim valuation | signed byte | -128..127 | max. Wertvolligkeit des OpCodes für Zielerreichung |
| avg aim valuation | signed byte | -128..127 | mittlere Wertvolligkeit d.OpC. für Zielerreichung |
| max_grad_aim_valuation | signed byte | -128..127 | max. Erhöhung der Zielerreichung gegenüber der kürzeren OpCodeKombination ohne diesen OpCode (KBT) |
| avg_grad_aim_valuation | signed byte | -128..127 | mittl. Erhöhung der Zielerreichung durch letzten OpC. |

Fig. 7

Datentypen: Boolean 1 Bit, BCD/Nibble 4 Bit, Byte/char(1) 8 Bit, Word/short 16 Bit, DWord/Integer 32 Bit, QWord/number(19) 64 Bit, number/number(38,0) 128 Bit (38 Digits ≙ 16 Bytes), varchar2(N) String variabler Länge aus max.N Character, long sehr langer String aus max(longDef) Character.

Die folgenden Kombinations-Tabellen werden dynamisch angelegt, haben die gleichen Non-PK-Columns, wie die OBT bzw. OLT bzw. ORT, jedoch für jede zusätzliche Code-Kombinations-Anzahl einen weiteren OpCode mehr im PK:

Kombinations-Register-Tabelle: [CRT(i), i = Anz.OpCodes, für die OpCode-Kombination, CRT(1)=ORT]

| Spalte: | Datentyp | Wertebereich | Bedeutung: |
|--|-------------|----------------------|---|
| OpCode 1 (PK) | integer | 0-2 ³² -1 | Opcode 1 (erster der Befehlskombination) |
| {for all OpCodes} (PK) | je integer | 0-2 ³² -1 | {für alle Opcodes 2 bis N-1} |
| OpCode N (PK) | integer | 0-2 ³² -1 | Opcode N (letzter der Befehlskombination) |
| IniConNr (PK) | signed byte | -31..30 | Anfangsbedingungs-Nr. |
| Register ID dest (PK) | signed byte | 0..127 | ein betroffenes Ziel-Register der Ausführung, s. RIT. |
| Register ID source (PK) | signed byte | -1..127 | -1 oder ein mögliches Quell-Register, siehe RIT. |
| {Gleiche Spalten, wie in der OpCode-Register-Tabelle.} | s.o. | s.o. | jede Kombinations-Register-Tabelle hat unter dem PK die gleichen Spalten, die die OpCode-Register-Tabelle auch hat. |

Fig.8

Kombinations-Lern-Tabelle: [CLT(i), i = Anz.OpCodes, für die OpCode-Kombination, CLT(1)=OLT]

| Spalte: | Datentyp | Wertebereich | Bedeutung: |
|--|-------------|----------------------|---|
| OpCode 1 (PK) | integer | 0-2 ³² -1 | Opcode 1 (erster der Befehlskombination) |
| {for all OpCodes} (PK) | je integer | 0-2 ³² -1 | {für alle Opcodes 2 bis N-1} |
| OpCode N (PK) | integer | 0-2 ³² -1 | Opcode N (letzter der Befehlskombination) |
| IniConNr (PK) | signed byte | -31..30 | Anfangsbedingungs-Nr. |
| {sonst gleiche Spalten, wie in der OpCode-Lern-Tabelle.} | s.o. | s.o. | jede Kombinations-Lern-Tabelle hat unter dem PK die gleichen Spalten, die die OpCode-Lern-Tabelle auch hat. |

Fig.9

Kombinations-Basis-Tabelle: [CBT(i), i = Anz.OpCodes, für die OpCode-Kombination, CBT(1)=OBT]

| Spalte: | Datentyp | Wertebereich | Bedeutung: |
|--|------------|----------------------|---|
| OpCode 1 (PK) | integer | 0-2 ³² -1 | Opcode 1 (erster der Befehlskombination) |
| {for all OpCodes} (PK) | je integer | 0-2 ³² -1 | {für alle Opcodes 2 bis N-1} |
| OpCode N (PK) | integer | 0-2 ³² -1 | Opcode N (letzter der Befehlskombination) |
| {sonst gleiche Spalten wie in der OpCode-Basis-Tabelle.} | s.o. | s.o. | jede Kombinations-Basis-Tabelle hat unter dem PK die gleichen Spalten, die die OpCode-Basis-Tabelle auch hat. |

Fig.10

CBT(max.) = CPT = Kombinations-Plan-Tabelle = Entstehungsort des Ergebnis-Programms.

Programmierziel- und Bewertungsfunktions-Tabellen:**Ziellösungs-Tabelle: [AST - Lösungen (aim-solution) aller gestellten Programmier-Aufgaben]**

| Spalte: | Datentyp | Wertebereich | Bedeutung: |
|-------------------------|----------|------------------------|---|
| Aim ID (PK) | short | 0..65535 | Identifiziert das Programmierziel |
| Solution Nr (PK) | byte | 0..255 | laufende Nr des Lösungsprogramms |
| aim Program | long | String | OpCode-Kombination des Lösungs-Prg. als String. |
| Program length | short | 1..65535 | Länge d. Lösungs-Prgs in Doublewords, aufgerundet |
| cycles of execution | integer | 1..2 ³² -1 | Ausführungszeit des Lös.-Prgs in Taktzyklen |
| used Registers BitCode | number | 1..2 ¹²⁸ -1 | Bitcode aller im Lös.-Prg. benutzten Register |
| used Operations Bitcode | number | 1..2 ¹²⁸ -1 | Bitcode aller im Lös.-Prg. benutzten Operationen |
| used aim Valuation Func | short | 0..65535 | Identifiziert die benutzten Zielnähe-Bewertungsfunktion |

Fig. 11

Ziel-Beschreibungs-Tabelle: [ADT - Zielprogramm-Beschreibung (aim-description) und Identifikation]

| Spalte: | Datentyp | Wertebereich | Bedeutung: |
|-----------------------------|--------------|------------------------|--|
| Aim ID (PK) | short | 0..65535 | Identifiziert das Programmierziel |
| aim Description | varchar2(32) | ≤32 Bytes | Beschreibung der Programmieraufgabe |
| used Processor Mode | integer | 0-2 ³² -1 | Flags über CCR Control-Register-Bits |
| all dest Register BitCode | number | 1..2 ¹²⁸ -1 | Bitcode aller Eingabe-Register der Aufgabe |
| all source Register BitCode | number | 1..2 ¹²⁸ -1 | Bitcode aller Ausgabe-Register der Aufgabe |
| unused Register BitCode | number | 1..2 ¹²⁸ -1 | Bitcode aller nicht zu benutzenden Register |
| unused Operation BitCode | number | 0..2 ¹²⁸ -1 | Bitcode aller mit Sicherheit nicht zu verwendenden Operationen (default = \$0000.0000:0000.0000) |
| aim implement solutions | long | String | String aus Aim ID's (words) früherer, hier einzubindender Lösungen. |
| aim fulfill valuation mode | boolean | 0 1 | Ziel-Bewertungsfunktion: 0=SQL ; 1=Maschinencode |
| aim fulfilled Flag Function | varchar2(99) | ≤99 Bytes | bool'sche Ziel-Erreicht Erkennungs-Funktion als String |
| aim Valuation FunctionID | varchar2(99) | ≤99 Bytes | Zielnähe-Bewertungs-Funktion als String |

Fig. 12

Funktion-Identifikations-Tabelle: [FIT - Tabelle der Bewertungsfunktions-Fragmente]
a.) für SQL-Funktionen:

| Spalte: | Datentyp | Wertebereich | Bedeutung: |
|----------------------|--------------|-----------------------|--|
| Function ID (PK) | signed byte | -1..127 | Identifikationsnummer der Teilfunktion |
| Function BitCode | number(19) | 0..2 ⁶⁴ -1 | BitCode dieser Teilfunktion |
| Function Name | char(5) | 5 Bytes | Funktions-Name |
| Function Type | byte | 0..99 | 0=value, 1=unitär, 2=binär, 3=ternär, ... |
| Function Flatten | signed byte | -127..127 | Funktions-Abflachungsgrad [pos=steiler, neg=flacher] |
| Function Template | varchar2(99) | ≤99 Bytes | SQL-Funktions-Template |
| Function Description | varchar2(99) | ≤99 Bytes | optionale Teilfunktions-Beschreibung |

Fig. 13a

| F.ID | Function BitCode | F.Name | F.T. | F.F. | Function Template | Function Description |
|------|------------------|--------|------|------|--|---|
| 0 | 1 | NUM | 0 | 0 | < FolgeWert > | es folgt eine Zahl |
| 1 | 2 | ENG_Y | 0 | 0 | ELT.energy_after | Energie nach Änderung |
| 2 | 4 | GRAD | 0 | 0 | ELT.energy_after -ELT.energy_before | Energie-Erhöhung |
| 3 | 8 | VALUE | 0 | 0 | CLT(n). <columnNr> | Wert aus dem Inhalt der folgenden Column-Nr |
| 4 | 16 | EREG | 0 | 0 | < EnergieRegister_ID > | ID des Energie-Registers |
| 5 | 32 | SGN | 1 | 0 | SIGN(%s) | Vorzeichen |
| 6 | 64 | ROUND | 1 | 0 | ROUND(%s, 0) | gerundet |
| 7 | 128 | INT | 1 | 0 | FLOOR(%s) | abgerundet |
| 8 | 256 | ABS | 1 | 0 | ABS(%s) | Betrag |
| 9 | 512 | NEG | 1 | 0 | -(%s) | Negation |
| 10 | 1.024 | ADD | 2 | 1 | ((%s) + (%s)) | Addition |
| 11 | 2.048 | SUB | 2 | -1 | ((%s) - (%s)) | Subtraktion |
| 12 | 4.096 | MUL | 2 | 4 | ((%s) * (%s)) | Multiplikation |
| 13 | 8.192 | DIV | 2 | -4 | ((%s) / (%s)) | Division |
| 14 | 16.384 | MOD | 2 | -2 | MOD(%s, %s) | Divisionsrest |
| 15 | 32.767 | SQRT | 1 | -8 | SQRT(%s) | QuadratWurzel |
| 16 | \$1.0000 | CBRT | 1 | -12 | POWER(%s, 1/3) | KubikWurzel |
| 17 | \$2.0000 | MIN | 2 | -10 | LEAST(%s, %s) | Minimum |
| 18 | \$4.0000 | MAX | 2 | -10 | GREATEST(%s, %s) | Maximum |
| 19 | \$8.0000 | LN | 1 | -48 | LN(%s) | Logarithmus naturalis |
| 20 | \$10.0000 | EXP | 1 | 48 | EXP(%s) | nat. Exponentialfkt. |
| 21 | \$20.0000 | LD | 1 | -32 | LOG(2, %s) | Logarithmus dualis |
| 22 | \$40.0000 | POT2 | 1 | 32 | POWER(2, %s) | 2-te Potenz von |
| 23 | \$80.0000 | SIN | 1 | -64 | SIN(%s) | Sinus |
| 24 | \$100.0000 | COS | 1 | -64 | COS(%s) | Cosinus |
| 25 | \$200.0000 | TAN | 1 | 127 | TAN(%s) | Tangens |
| 26 | \$400.0000 | ASIN | 1 | 127 | ASIN(%s) | ArcusSinus |
| 27 | \$800.0000 | ACOS | 1 | 127 | ACOS(%s) | ArcusCosinus |
| 28 | \$1000.0000 | ATAN | 1 | -127 | ATAN(%s) | ArcusTangens |
| 29 | \$2000.0000 | SINH | 1 | 40 | SINH(%s) | SinusHyperbolicus |
| 30 | \$4000.0000 | COSH | 1 | 50 | COSH(%s) | CosinusHyperbol. |
| 31 | \$8000.0000 | TANH | 1 | -127 | TANH(%s) | TangensHyperbol. |
| 32 | \$1.0000.0000 | LOG | 2 | -64 | LOG(%s, %s) | Logarithmus |
| 33 | \$2.0000.0000 | POT | 2 | 64 | POWER(%s, %s) | Potenzierung |
| 34 | \$4.0000.0000 | OR | 2 | 1 | ((%s) (%s)) | bitweises ODER |
| 35 | \$8.0000.0000 | AND | 2 | -1 | ((%s) & (%s)) | bitweises AND |
| 36 | \$10.0000.0000 | EQ | 2 | -127 | DECODE(%s, %s, 1, 0) | gleich |
| 37 | \$20.0000.0000 | LE | 2 | -127 | DECODE(GREATEST(%s - %s, 0), 0, 1, 0) | kleiner-gleich |
| 38 | \$40.0000.0000 | GE | 2 | -127 | DECODE(LEAST(%s - %s, 0), 0, 1, 0) | größer-gleich |
| 39 | \$80.0000.0000 | FRAME | 1 | -10 | GREATEST(LEAST(%s, +127), -128) | in signed-byte Rahmen: max. = 127, min. = -128 |
| 40 | \$100.0000.0000 | BITS | 1 | -64 | (1 & %s) + (2 & %s)/2 + (4 & %s)/4 + (8 & %s)/8 + | Anzahl Bits im vorigen Wert |
| 41 | \$200.0000.0000 | S_REG | 0 | 0 | ADT.all_source_Registers- BitCode | Quellregister-BitCode |
| 42 | \$400.0000.0000 | D_REG | 0 | 0 | ADT.all_dest_Registers BitCode | Zielregister-BitCode |
| 43 | \$800.0000.0000 | AIM_F | 0 | 0 | VAL(ADT.aim_fulfilled_Flag- Function) | Ergebnis der Ziel- erreichungsfunktion |
| ... | ... | ... | ... | ... | ... | ... |

Fig. 13b

b.) Für Maschinensprache-Funktionen:

Spalte: Datentyp Wertebereich Bedeutung:

| | | | |
|----------------------|--------------|------------------------|---|
| Function ID (PK) | signed byte | -1..127 | Identifikationsnummer der Teilfunktion |
| Function BitCode | number(19) | 0..2 ⁶⁴ -1 | BitCode dieser Teilfunktion |
| Operations BitCode | number | 0..2 ¹²⁸ -1 | BitCode der verwendeten OpCodes in dieser Funktion |
| Registers BitCode | number | 0..2 ¹²⁸ -1 | BitCode der verwendeten Register in dieser Funktion |
| Function Name | char(5) | 5 Bytes | Kurzbezeichnung der Teilfunktion |
| Function Type | byte | 0..99 | 0 = value, 1 = unitär, 2 = binär, 3 = ternär, ... |
| Function Flatten | signed byte | -128..127 | Funktions-Abflachungsgrad ($1 \triangleq f(x) = x$) |
| Function OpCodes | number | 1..2 ¹²⁸ -1 | Teilfunktion in Maschinensprache |
| Function Description | varchar2(99) | ≤99 Bytes | optionale Teilfunktions-Beschreibung |

Fig. 14a

| Func.ID | Func.BitCode | Oper.BitCode | Reg.BitCode | Func.Name | F.T. | Func.OpCodes | Function Descript. |
|---------|--------------|------------------------|-------------|-----------|------|--------------|------------------------------------|
| 0 | 1 | \$A000.4008 | < energy > | FRAME | 1 | s.u. Func.1 | Überläufe verhindern |
| 1 | 2 | \$28800.0009 | < energy > | SGN | 1 | s.u. Func.2 | Signum (Vorzeichen) |
| 2 | 4 | \$0000.0002 | < energy > | NEG | 1 | <NEG> | Negation |
| 3 | 8 | \$0000.0200 | < energy > | MUL2 | 1 | <SHLI> | Division durch 2 |
| 4 | 16 | \$0000.0400 | < energy > | DIV2 | 1 | <SHRI> | Multiplikation mit 2 |
| 5 | 32 | \$0000.0100: 4A00.8018 | (D0) (en) | ILOG2 | 1 | s.u. Func.3 | Logarithmus dualis |
| 6 | 64 | \$1000.C000: 0000.0000 | (FP0) (en) | ISQRT | 1 | s.u. Func.4 | Square-Root |
| 7 | 128 | | s. 1.4.2 | ICBRT | 1 | s.o. 1.4.2 | Cube-Root |
| 8 | 256 | \$0000.8000 | (en-1) (en) | MOV | 2 | <MOV> | Kopieren in Reg. vor Energy-Reg. |
| 9 | 512 | \$0000.8000 | (en-1) (en) | SWAP | 2 | s.u. Func.5 | Vertauschen mit Reg. vor Engy-Reg. |
| 10 | 1024 | \$0001.0000 | (en-1) (en) | ADD | 2 | <ADD> | Addition mit "-" |
| 11 | 2048 | \$0002.0000 | (en-1) (en) | SUB | 2 | <SUB> | Subtraktion "-" |
| 12 | 4096 | \$0004.0000 | (en-1) (en) | MUL | 2 | <MUL> | Multiplikation "-" |
| 13 | 8192 | \$0008.0000 | (en-1) (en) | DIV | 2 | <DIV> | Division "-" |
| ... | ... | ... | ... | ... | .. | ... | ... |

Fig. 14b

| Funktion: | OpCodes von: (Maschinencodeübersetzung u.g. Mnemonics, hier am Beispiel Motorola) |
|-----------|---|
| Func.1 | CMPI 127,(E) ; JLE (+2) ; MOVI #127,(E) ; CMPI -128,(E) ; JGE (+2) ; MOVI #-128,(E) |
| Func.2 | TST (E) ; JGE (+3) ; MOVI #-1,(E) ; JMP (+5) ; JGT (+3) ; MOVI #0,(E) ; JMP (+2) ; MOVI #+1,(E) |
| Func.3 | MOVI #31,D0 ; BTST D0,(E) ; JEQ (+3) ; DJMP D0,(-2) ; ADDI #1,D0 ; MOVE D0,(E) |
| Func.4 | FILD (E) ; FSQRT ; FIST (E) |
| Func.5 | MOVE (E-1),-(A7) ; MOVE (E),(E-1) ; MOVE (A7)+,(E) |

Fig. 14c

Bewertungs-Funktions-Tabelle: [VFT (valuation f.) - Tabelle der Bewertungsfunktionen]

| Spalte: | Datentyp | Wertebereich | Bedeutung: |
|----------------------------|--------------|--------------------|---|
| Valuation_Function_ID (PK) | signed short | ± 32767 | Identifizier der Bewertungsfunktionen (Energie neg.) |
| Valuation_Function_Type | char(1) | 'E' 'A' | 'E' = Energie-Bewertung, 'A' = Wertvolligkeit für Programmierzilerreichung, (ggf. später weitere) |
| Valuation_Function_Mode | boolean | 0 1 | 0 = SQL-Modus ; 1 = Maschinencode-Modus |
| Valuation_Function | varchar2(99) | ≤ 99 Bytes | Bewertungs-Funktion bzgl. Energie oder Zielerreichung |
| execution_counter | integer | $0 \dots 2^{32}-1$ | Anzahl der Funktions-Benutzungen |
| used_Functions_BitCode | number(19) | $0 \dots 2^{64}-1$ | BitCodes der verwendeten Teilfunktionen |
| Function_ID_Chain | varchar2(99) | ≤ 99 Bytes | Verkettung der Teilfunktionen (je Byte \triangleq Function ID) |
| avg_Func_execution_time | integer | $0 \dots 2^{32}-1$ | mittlere Ausführungszeit der Bew.Fkt. in Taktzyklen. |
| boundary_value_counter | integer | $0 \dots 2^{32}-1$ | Zähler f. Bewertungsergebnis = $-128 \dots +127$ |
| low_value_counter | integer | $0 \dots 2^{32}-1$ | Zähler f. Bewertungsergebnis in ± 16 |
| Valuation_Function_value | signed byte | $-128 \dots 127$ | Wertvolligkeit der Bewertungsfunktion = SAC.Self-Valuation Aim/Energy(Valuation Function, Values) |

Fig. 15a Initiale Einträge für Energie-Bewertung und Zielnähe-Bewertung:

| ID | Ty | M | Valuation Function |
|----|-----|---|---|
| -1 | 'E' | 0 | $\text{MAX}[\text{MIN}[\text{SGN}(\text{EnergyReg}' - \text{EnergyReg}^0) \cdot \text{SQRT}(\text{EnergyReg}' - \text{EnergyReg}^0) - 32 \cdot \text{CLT}(\text{i}).\text{Register_changed_BitCode} \& (1 \ll \text{Energy_Register_ID})], +127], -128]$ |
| 0 | 'A' | 0 | $\text{MAX}[\text{MIN}[16 \cdot \text{CLT}(\text{i}).\text{Register_changed_BitCode} \& \text{ADT.all_dest_Register_BitCode} + 16 \cdot \text{CLT}(\text{i}).\text{Register_source_BitCode} \& \text{ADT.all_source_Register_BitCode} + 32 \cdot \text{ADT.aim_fulfilled_Flag_Function}(\text{Aim_ID}) - \text{CLT}(\text{i}).\text{Processor_Mode_changed} - \frac{1}{4} \cdot \text{CLT}(\text{i}).\text{cycles_of_execution} - (\text{CLT}(\text{i}).\text{active} \text{inactive_ChkSum_corrupt}) - (\text{CLT}(\text{i}).\text{Exception_vect_changed} > 0) - (\text{CLT}(\text{i}).\text{Number_of_Exception} > 0) - \frac{1}{2}(\text{CLT}(\text{i}).\text{ObCode_length_or_jump} > 4 \text{ oder } \leq 0)], +127], -128]$ |

| ex# | used F. BitCode | Function ID chain | ex.T | bdy# | low# | F.Val |
|-----|-----------------|--|------|------|------|-------|
| 0 | \$189.0040.983B | 2,5; 2,15; 12; 4,22,3,11,35,40,1,32,12; 11; 39 | 0 | 0 | 0 | 0 |
| 0 | \$EE9.0001.3AAA | 3,11,42,35,1,16,12; 3,12,41,36,1,16,12,10; 43,1,32,10, 3,7,11; 3,16,1,5,13,11; 3,3,11; 3,5,11 3,5,5,10; 3,8,5,10; 3,9,1,0,37,11; 3,9,1,5,38,11; 39 | 0 | 0 | 0 | 0 |

Fig. 15b**Statuszeile Künstliches Bewußtsein: [SAC (artificial consciousness) - Statuswerte des KB-Programms]**

| Spalte: | Datentyp | Wertebereich | Bedeutung: |
|----------------------------|---------------|--------------------|---|
| Programm_StartDate | timestamp | Zeit+Dat. | Datum und Uhrzeit des Programmstarts. |
| actual_Processor_Mode | integer | $0 \dots 2^{32}-1$ | Flags über CCR Control-Register-Bits |
| actual_CPT_Index | byte | $1 \dots 255$ | CBT(max(i) = actual_CPT_Nr) = akt.CPT |
| CxT_counter | short | $1 \dots 65535$ | Anzahl des Aufbaus der dynamischen CxT-Tabellen |
| Aims_total | short | $1 \dots 65535$ | Anzahl der Programmierziele insgesamt |
| Aims_solved | short | $0 \dots 65535$ | Anzahl gelöster Programmieraufgaben |
| actual_Aim_ID | short | $0 \dots 65535$ | ID des aktuellen Programmierzies |
| Aim_Valuation_Mode | boolean | 0 1 | Modus der Zielerreichungs-Bewertungsfunktion 0 = SQL-Modus ; 1 = Maschinencode-Modus |
| Aim_Valuation_FunctionID | signed short | $0 \dots 32767$ | Aktuelle VFT.Valuation_Function_ID bzgl. Zielannäherungs-Bewertung |
| Aim_Self_Valuation_Func | varchar2(400) | max. 400 Zeichen | PL/SQL-Bewertungsfunktion bzgl. der Effizienz der Rahmen-Zielerreichungs-Bewertungsfunktionen |
| Energy_Valuation_Mode | boolean | 0 1 | Modus der Energiehandlungs-Bewertungsfunktion 0 = SQL-Modus ; 1 = Maschinencode-Modus |
| Energy_Valuation_Func_ID | signed short | $-1 \dots -32768$ | Akt. VFT.Valuation_Function_ID bzgl. Energie-Bewert. |
| Energy_Self_Valuation_Func | varchar2(400) | max. 400 Zeichen | PL/SQL-Bewertungsfunktion bzgl. der Effizienz der Energie-Bewertungsfunktionen |
| max_Valuation_Function | signed short | $0 \dots 32767$ | höchste ID aller Bewertungsfunktionen in der VFT. |
| min_Valuation_Function | signed short | $-1 \dots -32768$ | niedrigste ID aller Bewertungsfunktionen in der VFT. |

Fig. 16

Energie-Lern-Tabelle: [ELT - bewertet energiespezifische anfangsbedingungsabhängige Handlungen]

| Spalte: | Datentyp | Wertebereich | Bedeutung: |
|--------------------------|-------------|-----------------------|---|
| Energy_action (PK) | number | 0..2 ^{128,1} | max. 16 Byte OpCode-Kombination der Energie-Register verändernden Handlung. |
| IniConNr (PK) | signed byte | -31..30 | Anfangsbedingungs-Nr. |
| Energy before | integer | 0..2 ^{32,1} | Energie-Register vor dieser Handlung |
| Energy after | integer | 0..2 ^{32,1} | Energie-Register nach dieser Handlung |
| min Operations BitCode | number | 0..2 ^{128,1} | BitCode der wahrscheinlich benutzten Operationen. |
| max Operations BitCode | number | 0..2 ^{128,1} | BitCode aller möglicherweise benutzten Operationen. |
| Register changed BitCode | number | 1..2 ^{128,1} | BitCode der hierbei veränderten Register. |
| Register source BitCode | number | 1..2 ^{128,1} | BitCode der hierbei ausgelesenen Register. |
| used cycles of execution | short | 1..65535 | Ausführungszeit der energiespezifischen Handlung |
| Energy valuation | signed byte | -128..127 | Ergebnis der akt. VFT.Energy valuation Function |
| Valuation Function ID | byte | 0..255 | benutzte Energie-Bewertungsfunktion |

Fig. 17

Energie-Basis-Tabelle: [EBT - Auswertung der energiespezifischen Handlungen]

| Spalte: | Datentyp | Wertebereich | Bedeutung: |
|----------------------------|-------------|-----------------------|--|
| Energy_action (PK) | number | 0..2 ^{128,1} | max. 16 Byte OpCode-Kombination der Energie-Register verändernden Handlung. |
| Execution counter | byte | 0..255 | Anzahl der ELT-Einträge bis jetzt |
| FatalError_counter | byte | 0..255 | Anzahl der verursachten schweren Fehler: schwere Fehler entsprechen den Spalten 3-7 der Lern-Tabelle, außer wenn Number_of_Exception = Divide-Error oder Overflow. |
| low Error counter | byte | 0..255 | Anzahl der Divide-Error oder Overflow-Exceptions |
| avg Energy after | integer | 0..2 ^{32,1} | mittlerer Energie-Wert nach dieser Handlung |
| all_Reg_dest_BitCode | number | 0..2 ^{128,1} | \exists ELT.Register changed BitCode \vee ELT(OpCode) |
| cut_Reg_dest_BitCode | number | 0..2 ^{128,1} | ζ ELT.Register changed BitCode \vee ELT(OpCode) |
| all_Reg_source_BitCode | number | 0..2 ^{128,1} | \exists ELT.Register source BitCode \vee ELT(OpCode) |
| cut_Reg_source_BitCode | number | 0..2 ^{128,1} | ζ ELT.Register source BitCode \vee ELT(OpCode) |
| max Operation BitCode | number | 0..2 ^{128,1} | \exists ELT.max Operation BitCode \vee ELT(OpCode) |
| min Operation BitCode | number | 0..2 ^{128,1} | ζ ELT.min Operation BitCode \vee ELT(OpCode) |
| all Operation BitCode | number | 0..2 ^{128,1} | \exists ELT.min Operation BitCode \vee ELT(OpCode) |
| cut Operation BitCode | number | 0..2 ^{128,1} | ζ ELT.max Operation BitCode \vee ELT(OpCode) |
| max write value | integer | 0..2 ^{32,1} | Maximum aller geschriebenen Energie-Werte |
| min write value | integer | 0..2 ^{32,1} | Minimum aller geschriebenen Energie-Werte |
| avg write value | integer | 0..2 ^{32,1} | Mittelwert aller geschriebenen Energie-Werte |
| max write gradient | integer | 0..2 ^{32,1} | maximale Differenz des geänderten Energie-Werts |
| min write gradient | integer | 0..2 ^{32,1} | minimale Differenz des geänderten Energie-Werts |
| avg write gradient | integer | 0..2 ^{32,1} | durchschnittliche Differenz des geänderten Werts |
| equal value probability | signed byte | -128..127 | Wahrscheinlichkeit: Ergebnis immer gleich |
| avg Energy gradient | signed int | $\pm 2^31$ | mittlerer Energie-Gradient dieser Handlung |
| equal Gradient probability | signed byte | -128..127 | Wahrscheinlichkeit: Gradient immer gleich |
| avg cycles of execution | short | 1..65535 | Ausführungszeit der energiespezifischen Handlung |
| avg Energy valuation | signed byte | -128..127 | Ergebnis der akt. VFT.Energy valuation Function |
| Valuation Function ID | byte | 0..255 | benutzte Energie-Bewertungsfunktion |

Fig. 18

3.2 Flußdiagramm des KB-Programms:

3.2.1 CxT(i)-Wertezuweisungen:

ORT bzw. CRT(i):

| |
|--|
| ORT.Register_ID_dest := log ₂ (Bit(OLT.Register_changed_Mask), dessen Veränderung hier betrachtet wird) |
| ORT.Register_ID_source := Register_ID(C°), if ORT.calculation_code > 0, sonst -1. |
| ORT.value before change := value(Register_ID_dest), vor OpCode-Ausführung. |
| ORT.value after change := value(Register_ID_dest), nach OpCode-Ausführung. |
| ORT.gradient if signed := MAX[MIN[ORT.value after change - ORT.value before change, +127], -128] |
| ORT.gradient if unsigned := MAX[MIN[ORT.value after change - ORT.value before change, +127], -128] |
| <p>ORT.Operation_BitCode := 1·(Flags'≠Flags°)&&V'(V'=V°)&&[NF&&(V₁' < 0) ZF&&(V₁' = 0)]</p> <p>+ 2·[(V₁' = -V₁'°)&&V'(V'=V°)] + 4·[(V₁' = ~V₁'°)&&V'(V'=V°)] + 8·[(V₁' = 0LB)&&V'(V'=V°)]</p> <p>+ 16·[(V₁' = V₁'° + 0LB)&&V'(V'=V°)] + 32·[(V₁' = V₁'° - 0LB)&&V'(V'=V°)] + 64·[(V₁' = V₁'° · 0LB)&&V'(V'=V°)]</p> <p>+ 128·[(V₁' = V₁'° / 0LB)&&V'(V'=V°)] + 256·[(V₁' = V₁'° % 0LB)&&V'(V'=V°)] + 512·[(V₁' = V₁'° · 2° 0LB)&&V'(V'=V°)]</p> <p>+ 2¹⁰·[(V₁' = V₁'° / 2° 0LB)&&V'(V'=V°)] + 2¹¹·[(V₁' = V₁'° 0LB)&&V'(V'=V°)] + 2¹²·[(V₁' = V₁'° & 0LB)&&V'(V'=V°)]</p> <p>+ 2¹³·(Flags'≠Flags°)&&V'(V'=V°)&&[(ZF=1)&&(2° 0LB ~V°) (ZF=0)&&(2° 0LB V₁'°)]</p> <p>+ 2¹⁴·(Flags'≠Flags°)&&V'(V'=V°)&&[NF&&(V₁' < 0LB) ZF&&(V₁' = 0LB)] + 2¹⁵·[(V₁' = C₁'°)&&V'(V'=V°)]</p> <p>+ 2¹⁶·[(V₁' = V₁'° + C₁'°)&&V'(V'=V°)] + 2¹⁷·[(V₁' = V₁'° - C₁'°)&&V'(V'=V°)] + 2¹⁸·[(V₁' = V₁'° · C₁'°)&&V'(V'=V°)]</p> <p>+ 2¹⁹·[(V₁' = V₁'° / C₁'°)&&V'(V'=V°)] + 2²⁰·[(V₁' = V₁'° % C₁'°)&&V'(V'=V°)] + 2²¹·[(V₁' = 2° · V₁'°)&&V'(V'=V°)]</p> <p>+ 2²²·[(V₁' = V₁'° / 2°)&&V'(V'=V°)] + 2²³·[(V₁' = V₁'° C₁'°)&&V'(V'=V°)] + 2²⁴·[(V₁' = V₁'° & C₁'°)&&V'(V'=V°)]</p> <p>+ 2²⁵·(Flags'≠Flags°)&&V'(V'=V°)&&[(ZF=1)&&(2° C₁'° ~V₁'°) (ZF=0)&&(2° C₁'° V₁'°)]</p> <p>+ 2²⁶·(Flags'≠Flags°)&&V'(V'=V°)&&[NF&&(V₁' < C₁'°) ZF&&(V₁' = C₁'°)]</p> <p>+ 2²⁷·[(IP' ≤ IP°) (IP' > IP° + 4)]&&(Flags' = Flags°)&&V'(V'=V°)</p> <p>+ 2²⁸·[(IP' ≤ IP°) (IP' > IP° + 4)]&&(Flags' = Flags°)&&V'(V'=V°)&&(NF&VF !NF&VF) + ... VJcc(CCR)</p> <p>+ 2⁴⁰·[(IP' ≤ IP°) (IP' > IP° + 4)]&&(V₁' = V₁'° - 1) (V₁' = -1)&&(Flags' = Flags°)&&V'(V'=V°)</p> <p>+ 2⁴¹·[(IP' = IP° ± 0LB)&&(SP) = IP°)&&(Flags' = Flags°)&&V'(V'=V°)</p> <p>+ 2⁴²·[(IP' = -4(SP))&&(Flags' = Flags°)&&V'(V'=V°)] + 2⁴³·[(V₁' ≠ V₁'°)&&(I other_Integer_Operation_BitCode)]</p> <p>+ 2⁴⁴·[(V_F' ≠ V_F'°)&&(I other_FloatingPoint_Operation_BitCode)] + 2⁴⁵·[(CCR-Flags' = 0)&&V'(V'=V°)]</p> <p>+ 2⁴⁶·[(V₁' = C_F'°)&&V'(V'=V°)] + 2⁴⁷·[(V_F' = C₁'°)&&V'(V'=V°)] + 2⁴⁸·[(V_F' = V_F'° + C₁'°)&&V'(V'=V°)]</p> <p>+ 2⁴⁹·[(V_F' = V_F'° - C₁'°)&&V'(V'=V°)] + 2⁵⁰·[(V_F' = V_F'° · C₁'°)&&V'(V'=V°)] + 2⁵¹·[(V_F' = V_F'° / C₁'°)&&V'(V'=V°)]</p> <p>+ 2⁵²·(Flags'≠Flags°)&&V'(V'=V°)&&[NF&&(V_F' < C_F'°) ZF&&(V_F' = C_F'°)]</p> <p>+ 2⁵³·[(V_F' = 1.0) (V_F' = 0.0) (V_F' = π) (V_F' = e)]&&V'(V'=V°)</p> <p>+ 2⁵⁴·[(V_F' = -V_F'°)&&(V_F' < 0)&&V'(V'=V°)] + 2⁵⁵·[(V_F' = C_F'°)&&V'(V'=V°)]</p> <p>+ 2⁵⁶·[(V_F' = V_F'° + C_F'°)&&V'(V'=V°)] + 2⁵⁷·[(V_F' = V_F'° - C_F'°)&&V'(V'=V°)]</p> <p>+ 2⁵⁸·[(V_F' = V_F'° · C_F'°)&&V'(V'=V°)] + 2⁵⁹·[(V_F' = V_F'° / V_F'°)&&V'(V'=V°)] + 2⁶⁰·[(V_F' · V_F'° = V_F'°)&&V'(V'=V°)]</p> <p>+ 2⁶¹·[V_F' = sin(V_F'°)]&&V'(V'=V°) + 2⁶²·[V_F' = cos(V_F'°)]&&V'(V'=V°) + 2⁶³·[(V_F' = atan(V_F'°)]&&V'(V'=V°)]</p> <p>+ 2⁶⁴·[(V_F' = V_F'° · 2° · V_{F-1}'°)&&V'(V'=V°)] + 2⁶⁵·[(V_F' = V_{F-1}'° · log₂(V_F'°)&&V'(V'=V°)]</p> <p>+ 2⁶⁶·(Flags'≠Flags°)&&V'(V'=V°)&&[NF&&(V_F' < C_F'°) ZF&&(V_F' = C_F'°)] + 2⁶⁷·[(V₁' = C_S'°)&&V'(V'=V°)]</p> <p>+ 2⁶⁸·[(V_S' = C₁'°)&&V'(V'=V°)] + ..., mit V' = value_after_change (≠Flags), V° = value_before_change, C° = value(Register_ID_source). Es muß hierbei über alle Register_ID_source(Art) gechecked werden. Trotz gleichen Register-ID's im PK können mehrere Bits gesetzt werden [z.B. wg. 4 = 2 + 2 = 2° 2 = SHL(2) = ...]</p> |

Fig. 19

OLT bzw. CLT(i):

| |
|--|
| OLT.Processor_Mode_Changed := $\neg \{ \text{EFlags}_{\pi} / \text{SR}_u \ \& \ ! \exists 2^{\circ} \text{CCR_Flags} \} > 0 \ $ ORT.value after change(Register_ID eines Spezial-Registers) |
| OLT.aim_valuation := VFT.Aim_Valuation_Function(SAC.Aim_Valuation_FunctionID, ORT.xxxxx, Registers_changed_BitCode, Registers_source_BitCode, min_Operations_BitCode, max_Operations_BitCode, used_cycles_of_execution, ...) |
| CLT(n).gradient_aim_valuation := CLT(n).aim_valuation - CLT(n-1).aim_valuation |
| alle anderen Tabellenfeld-Zuweisungen sind anhand der OLT-Beschreibung in Fig.6 hinreichend erklärt. |

Fig. 20

OBT bzw. CBT(ii):

| |
|--|
| OBT.Execution_counter := Execution_counter + 1 |
| OBT.FatalError_counter := FatalError_counter + (0 < OLT.Number_of_Exception ≠ Devide_Error, Overflow) OLT.active_ChkSum_corrupt OLT.inactive_ChkSum_corrupt OLT.Exception_vect_changed OLT.Processor_Mode_changed) |
| OBT.Jump_longOp_probability := MAX[MIN[Jump_probability + (OLT.OpCode_length_or_jump ≤ 0) + (OLT.OpCode_length_or_jump > 4), +127], -128] |
| OBT.avg_OpCode_jump_length := (execution_counter * avg_OpCode_jump_length + akt.OpCode_jump_length) / (execution_counter + 1) |
| OBT.OpCode_len_unconfirmed := OpCode_len_unconfirmed (avg_OpCode_length ≠ akt.OpCode_length) |
| OBT.avg_cycles_of_execution := (execution_counter * avg_cycles_of_execution + akt.cycles_of_execution) / (execution_counter + 1) |
| OBT.exec_cycles_unconfirmed := exec_cycles_unconfirmed (avg_cycles_of_execution ≠ akt.cycles_of_execution) |
| OBT.Register_write_probability := MAX[MIN[Register_write_probability + 2*[(min.Reg.ID ≤ ORT.Column_ID_OLT ≤ max.Reg.ID) && ORT.value_before_change ≠ ORT.value_after_change] - 1, +127], -128] |
| OBT.Register_copy_probability := MAX[MIN[MIN(Register_copy_probability + 2*[(min.Reg.ID ≤ ORT.Column_ID_OLT ≤ max.Reg.ID) && ORT.value_before_change ≠ ORT.value_after_change && (min.Reg.ID ≤ ORT.Column_ID_source ≤ max.Reg.ID)] - 1, +127], -128] |
| OBT.Memory_write_probability := MAX[MIN[Memory_write_probability + 2*[(min.Adr.Reg.ID ≤ ORT.Column_ID_OLT ≤ max.Adr.Reg.ID) && ORT.value_before_change ≠ ORT.value_after_change] - 1, +127], -128] |
| OBT.Memory_copy_probability := MAX[MIN[Memory_copy_probability + 2*[(min.Adr.Reg.ID ≤ ORT.Column_ID_OLT ≤ max.Adr.Reg.ID) && ORT.value_before_change ≠ ORT.value_after_change && (min.Adr.Reg.ID ≤ ORT.Column_ID_source ≤ max.Adr.Reg.ID)] - 1, +127], -128] |
| OBT.Reg_to_Mem_probability := MAX[MIN[Reg_to_Mem_probability + 2*[(min.Adr.Reg.ID ≤ ORT.Column_ID_OLT ≤ max.Adr.Reg.ID) && ORT.value_before_change ≠ ORT.value_after_change && (min.Reg.ID ≤ ORT.Column_ID_source ≤ max.Reg.ID)] - 1, +127], -128] |
| OBT.Mem_to_Reg_probability := MAX[MIN[Mem_to_Reg_probability + 2*[(min.Reg.ID ≤ ORT.Column_ID_OLT ≤ max.Reg.ID) && ORT.value_before_change ≠ ORT.value_after_change && (min.Adr.Reg.ID ≤ ORT.Column_ID_source ≤ max.Adr.Reg.ID)] - 1, +127], -128] |
| OBT.Multi_Reg_write_prob := wie bei Register_write_probability, jedoch mit min.2 zutreffenden ORT.Column_ID_OLT-Einträgen. |
| OBT.Multi_Mem_write_prob := wie bei Memory_write_probability, jedoch mit min.2 zutreffenden ORT.Column_ID_OLT-Einträgen. |
| OBT.Multi_Reg_to_Mem_prob := wie bei Reg_to_Mem_probability, jedoch mit min.2 zutreffenden ORT.Column_ID_OLT+Column_ID_source-Einträgen. |
| OBT.Multi_Mem_to_Reg_prob := wie bei Mem_to_Reg_probability, jedoch mit min.2 zutreffenden ORT.Column_ID_OLT+Column_ID_source-Einträgen. |
| OBT.xxx_Reg_source dest BitCode: siehe Tabellenbeschreibung |
| OBT.xxx_calculation BitCode: siehe Tabellenbeschreibung |
| OBT.max_write_value := MAX(max_write_value, ORT.value_after_change) |
| OBT.min_write_value := MIN(min_write_value, ORT.value_after_change) |
| OBT.avg_write_value := (execution_counter * avg_write_value + ORT.value_after_change) / (execution_counter + 1) |
| OBT.max_write_gradient := MAX(max_write_gradient, ORT.value_after_change - ORT.value_before_change) |
| OBT.min_write_gradient := MIN(min_write_gradient, ORT.value_after_change - ORT.value_before_change) |
| OBT.avg_write_gradient := (execution_counter * avg_write_gradient + ORT.value_after_change - ORT.value_before_change) / (execution_counter + 1) |
| OBT.evaluated_source_[Num]Register := Wahrscheinlichkeitsfunktion(xxx_Reg_source_BitCode, confirmation_counter) |

| |
|--|
| OBT.evaluated dest Register[2] := Wahrscheinlichkeitsfunktion(xxx Reg_dest BitCode, confirm.ctr.) |
| OBT.evaluated Operation ID := Wahrscheinlichkeitsfunktion(xxx Operation BitCode, confirm.ctr.) |
| OBT.Confirmation_counter := Confirmation_counter + exist(äquivalenter OLT+ORTs-Eintrag mit niedrigerer IniConNr) |
| OBT.max_aim_valuation := MAX(max_aim_valuation, OLT.aim_valuation) |
| OBT.avg_aim_valuation := (execution_counter * avg_aim_valuation + OLT.aim_valuation) / (execution_counter + 1) |
| CBT(n).max_grad_aim_valuation := MAX(CBT(n).max_aim_valuation, CLT(n).aim_valuation) - CBT(n-1).max_aim_valuation. |
| CBT(n).avg_grad_aim_valuation := (execution_counter * CBT(n).avg_aim_valuation + CLT(n).aim_valuation) / (execution_counter + 1) - CBT(n-1).avg_grad_aim_valuation |

Fig.21

3.2.2 ELT und EBT -Wertezuweisungen:

| |
|---|
| ELT.max Operations BitCode := OLT.max Operations OpCode |
| ELT.min Operations BitCode := OLT.min Operations OpCode |
| ELT.Register changed BitCode := OLT.Registers changed BitCode |
| ELT.Register source BitCode := OLT.Registers source BitCode |
| ELT.Energy_Valuation := VFT.Energy_valuation_Function(SAC.Energy_Valuation_FunctionID, Energy_after, Energy_before, Registers_changed BitCode, Registers_source BitCode, min Operations BitCode, max Operations BitCode, used cycles of execution, ...) |
| ELT.Valuation_Function_ID := zur Berechnung von Energy_Valuation benutzte VFT.Valuation Function ID |
| EBT.avg_Energy_after := (execution_counter * avg_Energy_after + ELT.Energy_after) / (execution_counter + 1) |
| EBT.equal value probability := equal value probability + 2 * (avg_Energy_after = ELT.Energy_after) - 1 |
| EBT.avg_Energy_gradient := (execution_counter * avg_Energy_gradient + ELT.Energy_after - ELT.Energy_before) / (execution_counter + 1) |
| EBT.equal_gradient_probability := equal_gradient_probability + 2 * (avg_Energy_gradient = ELT.Energy_after - ELT.Energy_before) - 1 |
| EBT.xxx Operations Registers BitCode siehe Tabellenbeschreibung |
| EBT.avg_cycles_of_execution := (execution_counter * avg_cycles_of_execution + ELT.used_cycles_of_execution) / (execution_counter + 1) |
| EBT.avg_Energy_Valuation := (execution_counter * avg_Energy_Valuation + ELT.Energy_valuation) / (execution_counter + 1) |

Fig.22

3.2.3 Definitionen zum Lesen des Flußdiagramms:

- Anweisungen** kennzeichnet eine Anweisung oder eine Anweisungsfolge.
- Bedingung erfüllt ?** JA: verzweigt horizontal, NEIN: unten weiter.
- CodeFortführung** kennzeichnet eine Sprung-Marke zu bzw. von einem anderen Teil des Flußdiagramms.
- Anweisungs-Block** kennzeichnet einen Block bereits vorher definierter Anweisungen.

Im Flußdiagramm sind aufgrund der Aufwendigkeit nicht alle Details akribisch beschrieben, jedoch ist die Grundlage der Funktionsweise klar und verständlich dargelegt. Selbstverständliche Dinge, wie Cursor-Close, oder das mitfüllen nicht explizit erwähnter, aber vorhandener und keinen besonderen Algorithmus benötigender Tabellenfelder, gilt als selbstverständlich angenommen, da die Bedeutung der Tabellenfelder bereits unter 3.1.2 erklärt ist und deren Zuweisungen unter 3.2.1 bzw. 3.2.2.

Im Flußdiagramm bedeutet "Eintrag in der ORT generieren und OBT aktualisieren" einen Verweis auf die Wertezuweisungsalgorithmen in Fig.19-21.

Fig.23

3.2.5 KB-Flußdiagramm:

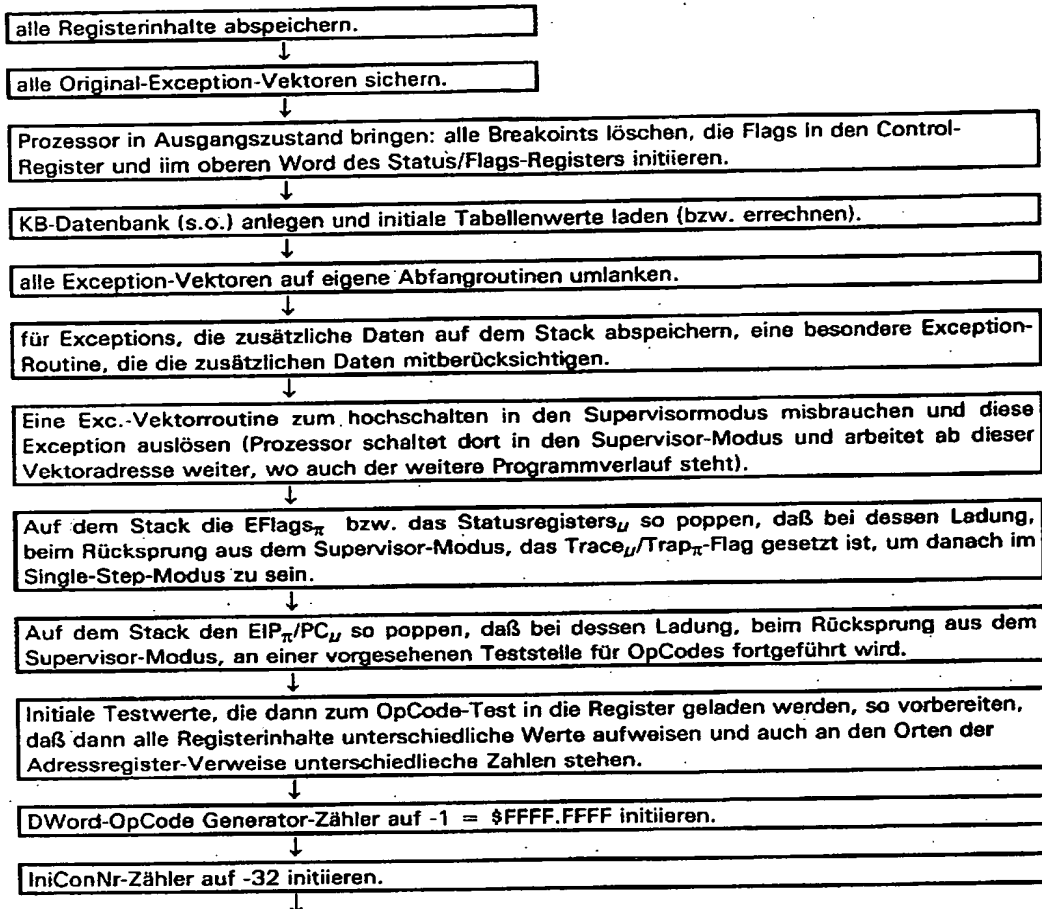
a.) Initiale Vorbereitungen:

Fig.24a

b.) Basis-Lernen:

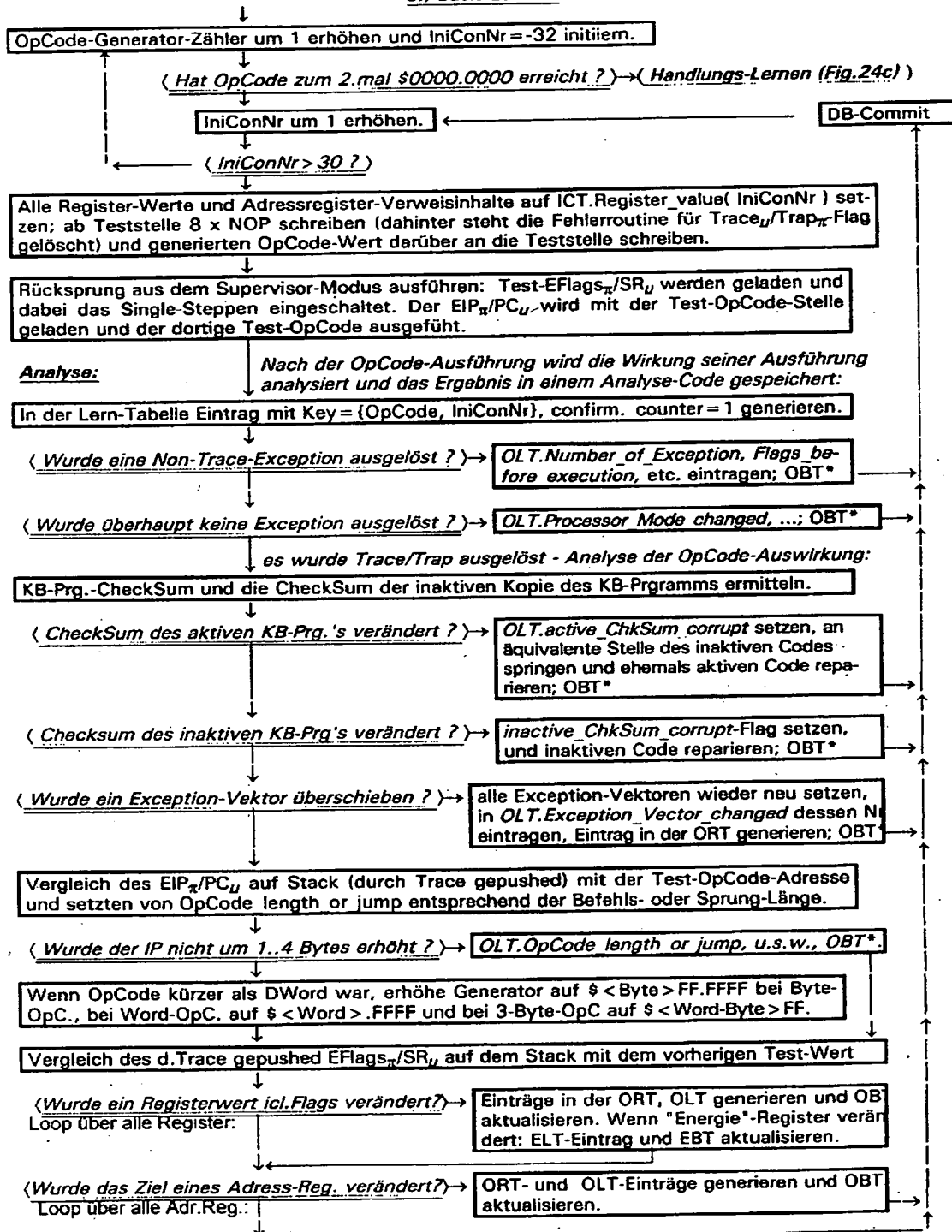


Fig. 24b

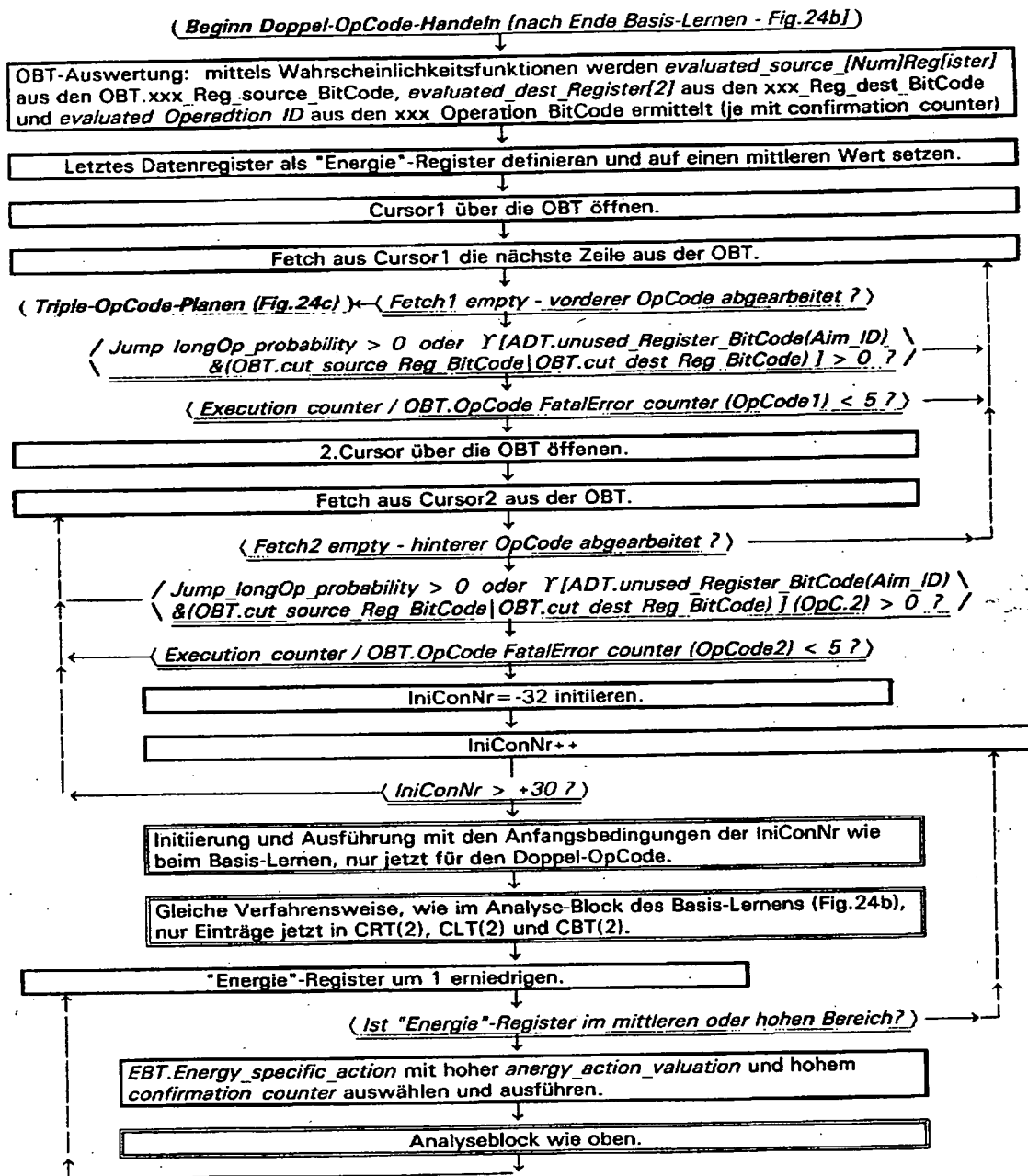
c.) Doppel-OpCode-Handeln:

Fig.24c

d.) Triple-OpCode-Planen:

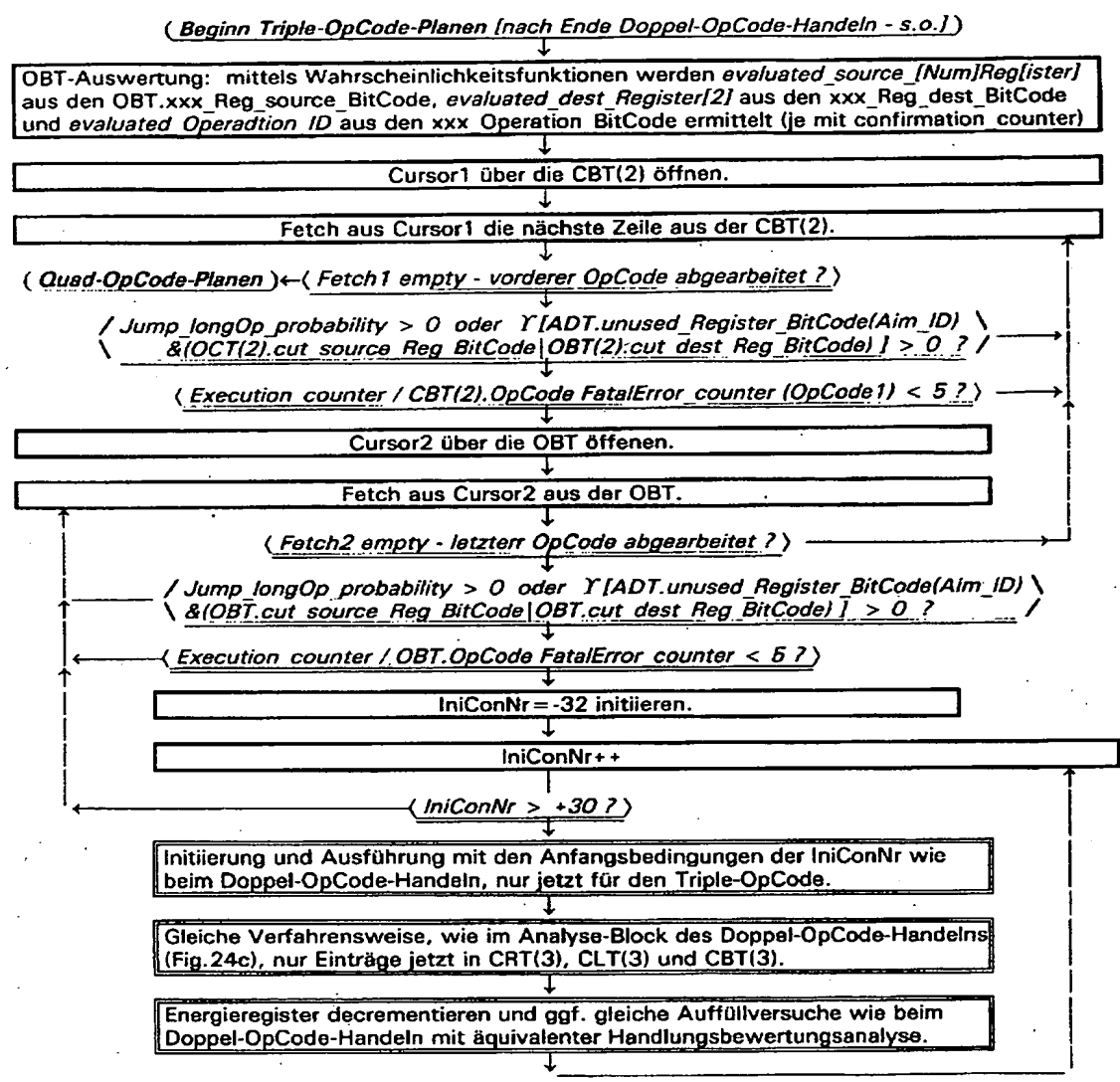


Fig. 24d

Verfahrensweise für höhere Kombinationen analog, mit CxT(n).

Letzte Seite Patentanmeldung vorab per Fax
G. Krüger